Pattern-Based Software Product Line Design using Role Modeling

Master's Thesis

Sven Schuster

November 20, 2014

Institute of Software Engineering and Automotive Informatics
Prof. Dr.-Ing. Ina Schaefer
at
Technische Universität Carolo-Wilhelmina zu Braunschweig

Supervision
Dipl.-Inf. Christoph Seidl
Technische Universität Dresden

Abstract

When developing large and complex software systems, software design is an essential topic. To establish high-quality design, various concepts emerged for individual software, resulting in *design patterns* as a means to reuse established solutions for recurring design problems. By addressing modularity and variability, specific design patterns allow customization of software. *Software product lines (SPL)* are an emerging concept to satisfy the demand for customization by reusing commonalities and variabilities within a family of similar software systems. *Feature-oriented programming (FOP)* is an implementation approach tailored to SPL development, modularizing realization artifacts along features and increasing variability-awareness. While design patterns are defined for individual software, their application in the context of SPLs is yet unknown. We argue that applying design patterns in SPLs in a decomposed manner could increase modularity and reusability across variants. With this thesis, we analyze this idea by conducting a family-based case study on the variability-aware application of design patterns in feature-oriented SPLs. To this end, we develop a variability-aware system representation based on the *Eclipse Modeling Framework (EMF)* as well as a family-based, automated design pattern detection technique. In the case study, we observed different design patterns being decomposed along features in a similar fashion. Generally, abstractions introducing the general pattern concepts are never decomposed, whereas concrete implementations often are. Using the results, we derive guidelines and application rules for specific design patterns in the context of SPLs. To describe the decomposed application of design patterns in a general fashion, we introduce *family role models (FRM)* as an extension to *role modeling*. We document our findings and derivations in a catalog of variability-aware design patterns.

Zusammenfassung

Bei der Entwicklung von großen und komplexen Softwaresystemen ist Softwaredesign ein wesentliches Thema. Verschiedene Konzepte entstanden für Individualsoftware, um ein hochwertiges Design zu entwickeln. *Entwurfsmuster* sind ein Mittel zur Wiederverwendung etablierter Lösungen für wiederkehrende Entwurfsprobleme. Durch Adressierung von Modularität und Variabilität ermöglichen bestimmte Entwurfsmuster eine Kundenanpassung von Software. *Softwareproduktlinien (SPL)* sind ein aufstrebendes Konzept, um die Nachfrage nach individueller Anpassung durch die Wiederverwendung von Gemeinsamkeiten und Variabilitäten innerhalb einer Familie von ähnlichen Softwaresystemen zu befriedigen. *Feature-orientierte Programmierung (FOP)* ist ein auf die Entwicklung von SPL zugeschnittener Implementierungsansatz, in dem Realisierungsartefakte anhand von Features modularisiert werden, was zu einem erhöhten Variabilitätsbewusstsein innerhalb der Realisierungsartefakte führt. Während Entwurfsmuster im Kontext von Individualsoftware definiert sind, ist ihre Anwendung im Kontext von SPL noch unbekannt. Wir behaupten, dass die dekomponierte Anwendung von Entwurfsmustern in SPL die Modularität und Wiederverwendbarkeit über Variantengrenzen hinaus erhöhen könnte. Mit dieser Arbeit analysieren wir diesen Aspekt mittels Durchführung einer familienbasierten Fallstudie über die variabilitätsbewusste Anwendung von Entwurfsmustern in Feature-orientierten SPL. Zu diesem Zweck entwickeln wir eine variabilitätsgewahre Systemdarstellung auf Basis des *Eclipse Modeling Framework (EMF)* sowie eine familienbasierte, automatisierte Detektionstechnik für Entwurfsmuster. In unser Fallstudie beobachteten wir unterschiedliche Entwurfsmuster, die in ähnlicher Weise entlang Features dekomponiert werden. Generell werden Abstraktionen, die das Grundkonzept des Musters einführen, nie dekomponiert, während konkrete Implementierungen oft dekomponiert werden. Anhand der Ergebnisse leiten wir Richtlinien und Anwendungsregeln für bestimmte Entwurfsmuster im Kontext von SPL ab. Um die Anwendung von Entwurfsmustern in einer allgemeinen Art und Weise zu beschreiben, führen wir *Familienrollenmodelle (FRM)* als Erweiterung der *Rollenmodellierung* ein. Wir dokumentieren unsere Ergebnisse in einem Katalog von variabilitätsgewahren Entwurfsmustern.

Contents

1 Introduction **10**
- 1.1 Motivation . . . 10
- 1.2 Research Goals . . . 11
- 1.3 Approach . . . 11
- 1.4 Contribution . . . 12
- 1.5 Structure . . . 12

2 Background **15**
- 2.1 Design Patterns . . . 15
  - 2.1.1 Pattern Categories by the Gang of Four . . . 17
  - 2.1.2 Pattern Structures and Relationships . . . 18
  - 2.1.3 Examples of Design Patterns . . . 19
- 2.2 Software Product Lines . . . 24
  - 2.2.1 Feature Models . . . 26
  - 2.2.2 Feature-Oriented Programming . . . 27
  - 2.2.3 Feature Interactions . . . 29
- 2.3 Role Modeling . . . 31
  - 2.3.1 Role Model Notation . . . 32
  - 2.3.2 Modeling Design Patterns using Role Modeling . . . 34
- 2.4 Metamodel-based Generative Software Development with EMF Ecore . . . 36
- 2.5 Previous Work . . . 38

3 Variability-Aware Design Patterns **43**
- 3.1 Motivation . . . 43
- 3.2 Approach . . . 44
- 3.3 Family Role Models . . . 46
- 3.4 Variability-Aware Design Pattern Catalog . . . 48
- 3.5 Summary . . . 49

4 FOPJaMoPP – FOP goes EMF **51**
- 4.1 Motivation . . . 51
- 4.2 JaMoPP – Java Model Parser and Printer . . . 51
- 4.3 Challenges of Parsing Feature-Oriented Java Code . . . 53
  - 4.3.1 Imports across Feature-Oriented Roles . . . 54
  - 4.3.2 Supertypes across Feature-Oriented Roles . . . 55
  - 4.3.3 Resolve Inter-Feature Element References . . . 56
  - 4.3.4 Resolve `original`-call . . . 58

4.4 Implementation . . . 58
4.4.1 Environment . . . 59
4.4.2 Architecture . . . 59
4.4.3 FOPJaMoPP Metamodel . . . 60
4.4.4 Workflow of FOPJaMoPP . . . 61
4.5 150% AST . . . 64
4.6 Summary . . . 65

5 Pattern Demon – Family-Based Design Pattern Detection 67
5.1 Motivation . . . 67
5.2 Automated Design Pattern Detection . . . 68
5.3 Approach . . . 69
5.3.1 Workflow . . . 70
5.3.2 Limitations . . . 73
5.4 Implementation . . . 73
5.4.1 EMF-IncQuery . . . 74
5.4.2 Architecture . . . 75
5.5 Detecting Design Patterns with Pattern Demon . . . 76
5.5.1 Common Pattern Rules . . . 76
5.5.2 Template Method . . . 80
5.5.3 Observer . . . 82
5.5.4 Strategy/Objectifier . . . 86
5.5.5 Composite . . . 88
5.6 Summary . . . 89

6 Case Study 91
6.1 Research Questions . . . 91
6.2 Experimental Setup . . . 91
6.2.1 Feature-Oriented Software Product Lines . . . 92
6.2.2 Design Patterns . . . 93
6.3 Methodology . . . 93
6.4 Results . . . 95
6.4.1 Composite . . . 95
6.4.2 Observer . . . 96
6.4.3 Strategy/Objectifier . . . 100
6.4.4 Template Method . . . 102
6.5 Discussion . . . 106
6.5.1 Implications of the Results . . . 106
6.5.2 Evaluating Pattern Demon . . . 109
6.6 Threats to Validity . . . 110
6.6.1 Construct Validity . . . 110
6.6.2 Internal Validity . . . 110
6.6.3 External Validity . . . 111

6.6.4 Reliability . . . 111
6.7 Summary . . . 111

7 Towards a Variability-Aware Design Pattern Catalog 113
7.1 Deriving Family Role Models . . . 113
7.1.1 Composite . . . 113
7.1.2 Observer . . . 113
7.1.3 Strategy / Objectifier . . . 114
7.1.4 Template Method . . . 114
7.2 Catalog Pages . . . 115
7.3 Summary . . . 119

8 Discussion 121
8.1 Limitations of Family Role Models . . . 121
8.2 Comparison to Previous Work . . . 122
8.3 Reasoning on Variability-Aware Application of Other Design Patterns . . . 122

9 Conclusion, Related and Future Work 125
9.1 Conclusion . . . 125
9.2 Related Work . . . 126
9.2.1 FOPJaMoPP . . . 126
9.2.2 Pattern Demon . . . 126
9.2.3 Variability-Aware Design Patterns . . . 126
9.3 Future Work . . . 127

List of Figures

2.1 Class diagram of template method pattern [16, p. 325] . . . 20
2.2 Class diagram of objectifier pattern [42] . . . 21
2.3 Class diagram of composite pattern [16, p. 164] . . . 21
2.4 Class diagram of observer pattern [16, p. 294] . . . 23
2.5 Printer example SPL . . . 25
2.6 Feature model for printer example SPL . . . 27
2.7 Toy example for feature-oriented implementation of printer example SPL . . . 29
2.8 FeatureHouse FSTs of feature-oriented implementation of class `InkjetPrinter` . . . 30
2.9 Illustration of derivative modules [23] . . . 31
2.10 Role model relations [33] . . . 33
2.11 Figure hierarchy example for role models from Riehle et al. [33] . . . 34
2.12 Role model example of observer pattern . . . 34
2.13 Mapping of observer role model to figure hierarchy example . . . 35
2.14 Detailed modeling of design patterns using role modeling . . . 36
2.15 Comparison of model-based and generative development . . . 37
2.16 Example Ecore metamodel of a library . . . 38
2.17 *Feature template method* in *GPL* (simplified) . . . 39
2.18 Example of decomposed visitor implementation in *GPL* . . . 41

3.1 Overview of the general approach of this work . . . 45
3.2 Notation for family role models . . . 47
3.3 Family role model for the strategy pattern . . . 48
3.4 Feature models extracts for variability-aware strategy pattern . . . 48
3.5 Catalog page for the strategy pattern as an example for the variability-aware design pattern catalog . . . 50

4.1 Simplified extract of JaMoPP metamodel . . . 52
4.2 Example of peculiarities in feature-oriented Java code . . . 53
4.3 Parsing FOP – Failing when collecting imports . . . 55
4.4 Parsing FOP – Failing when collecting supertypes . . . 56
4.5 FOPJaMoPP – Extensions to JaMoPP metamodel . . . 61
4.6 FOPJaMoPP – Workflow . . . 62
4.7 Extract of the 150% AST of the feature-oriented role `GPL.Vertex` of feature *BFS* in the GPL . . . 65

5.1 General Workflow of our family-based design pattern detection approach . . . 71
5.2 Minimal example for graph pattern matching on the observer pattern . . . 72

5.3 EMF-IncQuery query language example – Matching all tuples of classes that resemble a transitive extends relation . . . 74
5.4 EMF-IncQuery patterns: Methods and parameters . . . 77
5.5 EMF-IncQuery patterns: Checking relations between feature-oriented roles . . . 78
5.6 EMF-IncQuery patterns: Holding fields, arrays and lists . . . 79
5.7 Role model of the template method pattern . . . 80
5.8 EMF-IncQuery pattern specification for the template method pattern . . . 81
5.9 Role model of the observer pattern . . . 82
5.10 EMF-IncQuery pattern specification for the observer pattern . . . 83
5.11 Role model of the strategy pattern . . . 86
5.12 EMF-IncQuery pattern specification for the strategy pattern . . . 87
5.13 Role model of the composite pattern . . . 89
5.14 EMF-IncQuery pattern specification for the composite pattern . . . 89

6.1 Observer pattern instance `PlayListener` in *FeatureAMP* . . . 97
6.2 Extract of the feature model of *FeatureAMP* showing all features relevant to the observer patterns. . . . 98
6.3 Feature model extract of *GameOfLife* showing features relevant to observer and strategy patterns . . . 99
6.4 Observer pattern instance `ModelObserver` in *GameOfLife* . . . 100
6.5 Strategy pattern instance `GeneratorStrategy` in *GameOfLife* . . . 101
6.6 Template method pattern instance `DaemonThread` in *BerkeleyDB* . . . 103
6.7 Extract of the feature model of *BerkeleyDB* showing all features relevant to the template method pattern . . . 104
6.8 Template method pattern instance `AbstractNode` in *Violet* . . . 105
6.9 Extract of the feature model of *Violet* showing all features relevant to the template method pattern instance `AbstractNode` . . . 105

7.1 Legend for feature collaborations in family role models . . . 113
7.2 Catalog page for the composite pattern . . . 115
7.3 Catalog page for the observer pattern . . . 116
7.4 Catalog page for the strategy pattern . . . 117
7.5 Catalog page for the template method pattern . . . 118

List of Tables

2.1 Basic design patterns relevant for this work [16, 42] 19
2.2 Design patterns for typical software problems relevant for this work [16, 42] 22
2.3 Detected feature-oriented design patterns in prior work 40

6.1 Overview of the analyzed feature-oriented SPLs 92
6.2 Absolute numbers of detected design patterns 96

1 Introduction

In this chapter, we provide a brief introduction to this thesis. We start with a brief motivation in Section 1.1. Next, we present the research goals of this thesis in Section 1.2. We explain our approach in Section 1.3. After stating the contribution in Section 1.4, we finally provide the structure of this thesis in Section 1.5.

1.1 Motivation

Software design is a crucial task during software engineering. Various design goals emerged, including modularity, extensibility and reusability[1], aiming at developing high-quality software. In order to fulfill these goals, different design principles and concepts emerged in the context of *object-oriented programming (OOP)* [14, 10], resulting in *design patterns* as a means to reuse established solutions to common design problems [16, 42, 15, 8, 14]. Many design patterns focus on encapsulating variation to achieve modularity and reusability. Hence, specific design patterns have been applied to allow customization of software [2].

In recent years, due to an increasing demand for customizing software to the needs of a specific stakeholder, the idea of *software product lines (SPL)* gained momentum [11, 27]. SPLs reuse commonalities and variabilities to realize such customization while decreasing cost and effort. New implementation approaches tailored to SPL development, and categorized as annotative and compositional [4, 2] as well as transformational [35] approaches, have been introduced in order to increase variability-awareness along realization artifacts. Generally, these approaches are language-independent, however, most approaches are based on OOP.

Feature-oriented programming (FOP) [28, 6] is a compositional approach for SPL development that allows modularizing realization artifacts along *features*, where a feature is usually defined as an increment to functionality [4]. According to the needs of a stakeholder, i.e., a specific feature selection, realization artifacts of the selected features are composed to form a particular *variant* of the SPL. Due to its modular nature, FOP offers a new layer of design that allows refining realization artifacts and, thus, extending them with new functionality. However, only little is known about realizing high-quality design in the context of FOP [3, 22].

Even though design patterns are documented in extensive catalogs, they are only defined in the context of individual software systems but not for SPLs. Since various design patterns focus on reuse, modularity and encapsulating variability [16] and FOP realizes modularity in order to increase reusability and variability, we reasoned in prior work that applying design patterns in the context of FOP might affect the realization of variability mechanisms of FOP and vice versa [36]. Exploiting both variability mechanisms of FOP and design patterns could be beneficial, such that we might be able to observe a symbiotic relationship between both concepts. In preliminary studies, we analyzed feature-oriented SPLs for their use of design patterns and revealed the existence of design patterns

[1] Software Quality Standard (ISO/IEC 9126)

that are implemented in a decomposed fashion along features [37]. However, we were only able to analyze their variability-aware application to some limited extent.

1.2 Research Goals

While we observed that design patterns are applied in a decomposed and, thus, in a variability-aware fashion, only little is known about their actual variability-aware implementation in the context of FOP. Moreover, the impact of design patterns on how modularity is realized using FOP is unknown. In order to analyze this, we formulate the following research goals.

Research Goal 1 *(RG1).* *Reveal the variability-aware application of design patterns in the context of FOP.*
With this work, we aim at revealing how exactly design patterns are applied in the context of FOP, hence, how their implementation is decomposed along features. Because design patterns are a reference of established solutions for common design problems, we cannot reason on their variability-aware implementation, but have to collect evidence on their existence and their exact application. To obtain this information, we need to conduct a case study on existing feature-oriented SPLs, analyzing the decomposed application of design patterns.

Research Goal 2 *(RG2).* *Document guidelines and application rules in a catalog of variability-aware design patterns.*
Design patterns have been introduced by Gamma et al. [16] by documenting established solutions in an extensive catalog. Based on this design pattern catalog, we aim at formulating a similar catalog for variability-aware design patterns, using the results of the case study to derive guidelines and application rules. Because design patterns are informal descriptions of general solutions, we require a means to generally describe the variability-aware application of design patterns.

1.3 Approach

To realize the research goals of this work, we pursue the following approach.

Realizing RG1. We conduct a case study to reveal the variability-aware application of design patterns. To this end, we require an automated design pattern detection technique because manual detection is not suitable to obtain reliable results. Moreover, in order to conduct the pattern detection, we need a system representation of the whole product line, containing information on variability. Besides the pattern detection technique that we applied in our prior study [37], which was only successful to some limited extent, no approaches on automated design pattern detection in the context of SPLs exist. Moreover, existing system representations are not convenient for family-based detection. Hence, in this work, we develop both, a variability-aware system representation for feature-oriented SPLs, as well as an automated, family-based design pattern detection technique.

Realizing RG2. In order to create a design pattern catalog, we require a means for the general description of variability-aware design patterns. Role modeling [30, 33] is suitable for general descriptions by focusing on mere collaborations instead of a definite design. Because role modeling has successfully been applied to design patterns [31, 32], we employ role modeling for describing design patterns and extend the notion by means to describe feature distribution

and collaborations of features. Using the results of our case study, we derive guidelines and application rules for a variability-aware application of design patterns, leading to a variability-aware design pattern catalog.

1.4 Contribution

Realizing our research goals leads to the following contributions of this work.

1. *We propose the idea of variability-aware design patterns.*
   With variability-aware design patterns, we address standard design patterns for individual software applied in a decomposed fashion along features in the context of FOP.

2. *We introduce family role models (FRM) as a means to describe the variability-aware application of design patterns in a general fashion.*
   Using role modeling as a means to create general descriptions of collaborations instead of a definite design, we describe the decomposition of design patterns. We extend the notion of role modeling to also express collaborations of the corresponding features involved in realizing a specific design pattern instance. Because different feature models can be semantically equivalent, we use FRMs to depict the general collaborations of features instead of a definite feature model.

3. *We create* FOPJaMoPP*, a variability-aware system representation for feature-oriented* Java *code.*
   For the automated design pattern detection, we create a system representation for feature-oriented code that takes feature semantics and, thus, variability information into account.

4. *We develop* Pattern Demon*, a family-based design pattern detection technique for feature-oriented SPLs.*
   In order to conduct a case study on the application of design patterns in feature-oriented SPLs, we develop a family-based detection approach based on FOPJaMoPP.

5. *We create a catalog of variability-aware design patterns.*
   As the main contribution of this work, using the results of the case study, we create FRMs for the analyzed design patterns, derive guidelines and application rules for the implementation of design patterns in the context of FOP and create a catalog pages for variability-aware design patterns.

1.5 Structure

The remainder of this thesis is structured as follows. In Chapter 2, we provide necessary background information on design patterns, SPLs in general and FOP in particular, role modeling, metamodel-based generative software development as well as previous work on this topic. Next, in Chapter 3, we introduce the idea of variability-aware design patterns by presenting our overall approach and the ideas of family role models and the variability-aware design pattern catalog. We present FOPJaMoPP, the variability-aware system representation for feature-oriented Java code in Chapter 4. In Chapter 5, we present Pattern Demon, the family-based, automated design pattern detection technique. In Chapter 6, we employ both, FOPJaMoPP and Pattern Demon, for the case study on the variability-aware application of design patterns in existing feature-oriented SPLs. Next, in

Chapter 7, we present the main contribution of this work, by deriving family role models and creating catalog pages for variability-aware design patterns including guidelines and application rules. We discuss and reason about different issues and aspects that arose during the course of this thesis in Chapter 8. Finally, we give a brief conclusion on this thesis and state related and future work in Chapter 9.

2 Background

In this chapter, we provide the necessary background information for the following chapters. First, we describe object-oriented design patterns in Section 2.1. Afterwards, in Section 2.2, we describe the concept of software product lines, including feature-oriented programming, the implementation paradigm we are using in this thesis. Because we base our research on role modeling, we explain the idea behind modeling roles instead of classes and objects in Section 2.3. Since, in the following chapters, we also describe our effort of implementing the concepts needed for this work, we introduce the employed implementation approach of metamodel-based generative software development using EMF Ecore in Section 2.4. Finally, in Section 2.5, we provide information on our previous work concerning design patterns in feature-oriented product lines.

2.1 Design Patterns

Software development consists of many different and challenging tasks, including software design and implementation. During design and implementation, a lot of key aspects have to be considered, which mainly contribute to the quality of the respective software, hence, they are essential goals to achieve. These aspects are, amongst others[1]:

Modularity
: The software encompasses independent components, which therefore can be implemented and tested in isolation before integration.

Extensibility
: The software can be easily extended by new functionality without major changes to the existing system.

Maintainability
: Functionalities of the software can be easily modified, and bugs fixes are easy to accomplish.

Reusability
: Parts of the software can be easily reused to introduce new functionality without major changes.

In software design, it is crucial to address these aspects in order to develop a system that can withstand evolution such that developers can deal with modifications and further development without much effort. Especially these four aspects correlate with the quality of the design by directly regarding the implementation level of the software. In order to address these aspects, a variety of fundamental design concepts have evolved, such as *abstraction*, *encapsulation* or *information hiding* [10]. These concepts directly correlate with the aforementioned aspects of software quality. Using encapsulation and information hiding, for instance, developers can create independent subsystems

[1] Software Quality Standard (ISO/IEC 9126)

or modules, which can easily be reused. Moreover, combined with abstraction, maintainability and extensibility are increased because of loose coupling, interchangeability of such subsystems and easier understanding of the system.

Object-oriented programming (OOP) gained momentum in die 1960s and emerged to be a widely used programming paradigm, offering inheritance and object composition as concepts for encapsulation, modularity and reuse. Since then, various design principles have evolved, aiming at realizing these design concepts in order to fulfill the aforementioned key aspects [14]. Examples for such design principles are:

Dependency Inversion Principle
: Classes should depend upon abstractions rather than concrete classes.

Open Closed Principle
: Classes should be "open for extension, but closed for modification", thus extensible, but not modifiable.

Hollywood Principle
: Delegating control flow to dynamically registered components leads to loose coupling, which increases reusability and maintainability.

Principle of Least Knowledge
: Classes should not interact with too many other classes, thus realizing loose coupling.

Such design principles lead to software that fulfills the key aspects of software design. However, during software design, it is common that various recurring design problems emerge. For instance, objects depend upon the state of another object, or algorithms should be interchangeable at runtime. Such problems have to be solved without decreasing, and possibly while increasing, maintainability, extensibility and reusability. Hence, for example, we want to avoid introducing new dependencies to other classes or subsystems.

To this end, Gamma et al. [16], called the *Gang of Four (GoF)*, introduced *object-oriented design patterns* in 1995, offering general solutions to such common problems. However, Gamma et al. [16] did not invent these patterns, instead they documented existing and reliable design experience of software developers. Design patterns can be seen as best practices for specific situations [9]. Design patterns follow the mentioned object-oriented design principles in order to describe general solutions to specific problems.

According to Gamma et al. [16], a pattern consists of four essential parts:

Pattern name
: Giving names to design patterns and to the problem, the solution and the consequences, improves understanding and communication.

Problem
: The problem describes the application domain, the motivation and the intent of the pattern. A specific design problem (i.e., interchangeable algorithms), specific object structures (i.e., for inflexible design) or conditions to be met might be described.

Solution

The solution describes the structure (e.g., classes, objects and their relations with a class diagram) and, if necessary, the interaction between classes and objects (e.g., with a sequence diagram), providing a general template to be used in different situations.

Consequences

In the consequences part, *the results and trade-offs of applying the pattern* are discussed [16]. This includes the costs, benefits, dependencies and limitations. The costs include trade-offs in time and space, which are basically costs in time and memory consumption. Benefits may include an increased reusability of classes or whole (sub-)systems or an increased variability or maintainability. Moreover, dependencies describe which preconditions have to be met, and limitations capture difficulties for implementation or situations for which the pattern is not applicable.

Applying a design pattern does not simply mean to copy the proposed design solution of the pattern, but rather to adapt it to a specific problem. Therefore, a design pattern does not represent an enclosed design decision, but rather a general description of how classes and objects have to be related and how they have to interact to solve a design problem.

Moreover, applying design patterns to software is not a guarantee for a high-quality design, since design patterns also come with drawbacks, for instance, a high number of small classes or a lot of indirection. Design patterns are merely means to achieve a high-quality design when applied in a sensible and responsible manner [16].

Beyond mere design benefits, design patterns also provide advantages regarding the accessibility of the code for other developers resulting from well-structured design decisions. Additionally, because these design decisions are documented and named, design patterns improve and simplify the communication between developers [16, 9].

Gamma et al. [16] classified design patterns by their purposes into three categories, which we briefly overview in the following. Moreover, there has been a lot of ongoing research concerning design patterns, for instance, analyzing the patterns described by Gamma et al. [42], documenting new patterns such as *Objectifier* or *Role Object* [8, 15, 42] or applying design patterns [9]. Because they are relevant for this work, we concentrate on pattern structures and relationships described by Zimmer [42] in Section 2.1.2. Finally, we give an overview over patterns that are relevant for this work and exemplary illustrate some of them.

2.1.1 Pattern Categories by the Gang of Four

Gamma et al. [16] classified their design patterns by intent and the problem they address into the following three categories of *creational*, *structural* and *behavioral* patterns:

Creational Patterns. Creational patterns deal with the instantiation of classes. By hiding the instantiation process, creational patterns provide variability in any aspect regarding the creation of objects. The instantiation may vary in *which* object to create, *when* and *how* to create it and *where* to create it. There are two types of creational patterns. *Class creational patterns* use inheritance to flexibly change the class to be instantiated. *Object creational patterns* delegate the instantiation to another object.

An example for creational patterns is the *Abstract Factory*, where an interface is provided for creating similar or dependent objects without depending on their concrete subclass [16, p. 87 ff.]. With a *Singleton*, a developer can enforce that a class can only be instantiated once for the whole software system [16, p. 127 ff.].

Structural Patterns. Structural patterns address the composition of classes and objects in order to create large and variable structures. Using inheritance, *structural class patterns* describe ways to compose interfaces or implementations, while *structural object patterns* exploit the ability to replace composed objects at runtime, gaining flexibility and new functionality.

An example for structural patterns is the *Adapter*, where an interface of a class is converted into an interface, which a client expects, thus, structurally unifying them [16, p. 139 ff.]. With the *Composite* pattern, for instance, a hierarchical tree structure of objects can be composed [16, p. 163 ff.].

Behavioral Patterns. Behavioral patterns concentrate on variability at runtime, dealing "*with algorithms and the assignment of responsibilities between objects*" [16]. Rather than just providing collaborations of objects and classes like creational and structural patterns, behavioral patterns also describe the communication between them. *Behavioral class patterns* use inheritance to delegate behavior to different classes. *Behavioral object patterns* use object composition to distribute behavior between objects.

An example for behavioral patterns is the *Observer*, where a one-to-many relation of objects, that depend on another object's state, is defined [16, p. 293 ff.]. The *Strategy* pattern encapsulates each algorithm of a family of algorithms independently and makes them interchangeable at runtime [16, p. 315 ff.]. Similar to the Strategy pattern, the *Template Method* pattern defers steps of an algorithm, defined in a skeleton, to subclasses, resulting in interchangeable behavior [16, p. 325 ff.].

2.1.2 Pattern Structures and Relationships

Already Gamma et al. [16] noticed and documented relationships between their design patterns. Therefore, they mention related patterns for each pattern they describe. Also, they provide an overview in form of a map where they illustrate the pattern relationships and briefly outline each relation.

Although design patterns only describe a general approach to solve a problem instead of providing a complete solution, design patterns still have quite distinguishing, typical structures of how they are applied in object-oriented systems. Gamma et al. express these structures using class diagrams.

In contrast to Gamma et al., who describe the relationships between patterns addressing different issues such as the problem definition or specific implementation details, Zimmer [42] categorized the patterns on a completely problem-based level. Thus, Zimmer [42] revised the relationships between design patterns and classified them into three categories:

Pattern X uses pattern Y in its solution
: When applying pattern X, a subproblem is similar to the problem addressed by pattern Y.

Pattern X is similar to pattern Y
: Pattern X and Y address a similar kind of problem.

Pattern X can be combined with pattern Y
: A typical combination of patterns X and Y is known.

Design Pattern	GoF-Category	Description
Adapter	Structural	Transform interface of a class to another interface
Composite	Structural	Compose objects hierarchically into tree structures
Decorator	Structural	Dynamically attach additional responsibilities to objects
Facade	Structural	Provide unified interface for a whole subsystem
Mediator	Behavioral	Provide object to mediate the interaction of set of objects
Objectifier	None [42]	Objectify similar behavior in additional classes
Proxy	Structural	Provide a surrogate for an object to control access to it
Template Method	Behavioral	Define skeleton for algorithm & defer steps to subclasses

Table 2.1: Basic design patterns relevant for this work [16, 42]

Especially relationships of the first category, patterns using other patterns, occur remarkably often, since 12 of the 23 considered GoF-patterns use other patterns in their solutions. From these relations, Zimmer [42] derived the following layers of design patterns:

Basic design patterns and techniques
: Design patterns that are used by other patterns or that do not use other patterns themselves (cf. Table 2.1).

Design patterns for typical software problems
: General patterns that use basic patterns in their solutions and do not address any specific application domain (cf. Table 2.2).

Design patterns specific to an application domain
: Very specific patterns that are tailored to a specific domain. In this layer, only the *Interpreter*, which is used to parse simple languages, is mentioned, which we do not consider in this work [16, p. 243 ff.].

In the following subsection, we describe these categories in more detail by giving examples for them. As an addition to the patterns documented by Gamma et al. [16], more patterns have been documented in recent years, for example, the *extension object* or *role object* patterns [15, 8].

2.1.3 Examples of Design Patterns

In this section, we provide a few examples of important and common patterns to give an impression on how patterns are described, how they are applied and how some of them structurally depend on each other. In Table 2.1 and Table 2.2, we provide a short overview of the patterns that are important for this work, while giving a brief description of these patterns and outlining their usage-relationships.

Basic Design Patterns

In the following, we exemplary describe some of the basic design patterns, their intents and their relationships to other patterns. In Table 2.1, we list the basic design patterns identified by Zimmer [42] that are relevant for this work.

Template Method. One basic behavioral class pattern is the *template method* [16, p. 325 ff.]. Using inheritance, with this pattern the developer gains the ability to delegate algorithms or parts of algorithms to subclasses. A skeleton of an algorithm is defined in an operation of the parent class.

In certain situations, it might not be possible to specify every step of an algorithm because it may have to differ depending on the context. If steps of an algorithm have to be predefined nevertheless, an abstract skeleton of the algorithm can be described. The specification of context-dependent parts of the algorithm is then delegated to subclasses. In Figure 2.1, we illustrate this pattern. In the `AbstractClass`, we define the skeleton of an algorithm within a so-called *template method* by calling primitive operations. Of course, invariant parts can be specified within the `AbstractClass` or within other classes. We then specify the abstract primitive operations in a `ConcreteClass`.

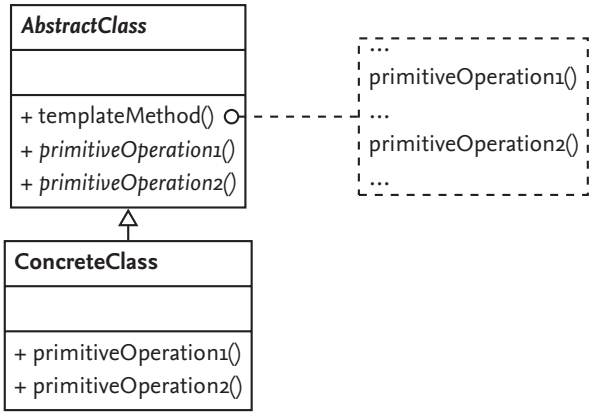


Figure 2.1: Class diagram of template method pattern [16, p. 325]

Using the template method, invariant code can be reused, which avoids code duplication, while variable code can be delegated to subclasses. Abstract primitive operations have to be specified within subclasses. Also, so called "hook" operations can be introduced, which provide default behavior and may be overridden by subclasses. By introducing abstraction and delegating some behavior to operations made concrete in subclasses, the template method relies on the aforementioned Hollywood principle, leading to variable algorithms that behave as the context requires.

Objectifier. The *objectifier* is a basic behavioral object pattern, which has been introduced by Zimmer [42] as a generalization to the later described *strategy* pattern. The basic idea of an objectifier is to encapsulate and, thus, objectify similar, yet varying behavior in additional classes making it interchangeable at runtime. An instance of such an additional class represents a specific behavior.

The objectifier tackles the frequently faced problem of separating the abstraction from its implementation and the interchange of implementations. In Figure 2.2, we illustrate the objectifier pattern with a class diagram. Here, the `Objectifier` interface is referenced by a `Client` calling the `operation` on the `Objectifier`. This call is delegated to one of its concrete implementations holding the concrete, desired behavior, depending on which instance the `Client` holds.

As a common and very basic pattern, the objectifier is the foundation of many other patterns including, amongst others, the strategy, observer, bridge or visitor patterns, all of them objectifying different behavior. Moreover, it is quite similar to the template method pattern, but differs in not providing a skeleton of an algorithm but rather encapsulating the whole behavior that is available to the client.

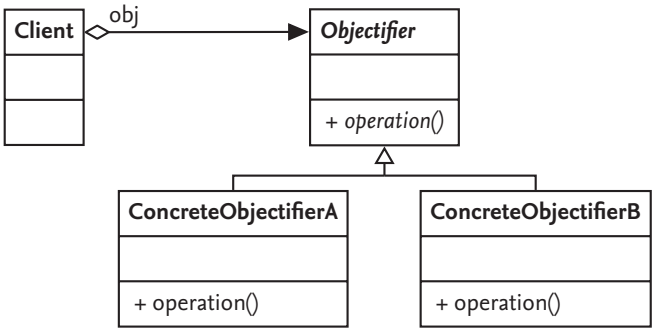


Figure 2.2: Class diagram of objectifier pattern [42]

Composite. Another basic design pattern is the *composite*, which provides the developer with the ability to compose objects in a hierarchical tree-structure to represent part-whole hierarchies [16, p. 163 ff.]. Usage of the composite pattern lets clients treat individual objects as well as compositions of objects uniformly.

In Figure 2.3, we depict a class diagram of the composite pattern. In order to realize the composite pattern, we define a `Component`, which declares all the operations that can be invoked on components, thus, on individual objects and compositions of objects. We define a subclass `Leaf`, which is used to handle individual objects. Additionally, in order to be able to also treat compositions of objects as components, we define a class `Composite` that provides access to a sequence of objects of type `Component`, which are the children of this component. Components can be added and removed from this sequence. This way, we can create a root element of type `Composite`, holding a number of objects of type `Component`, which can be individual components (`Leaf`) as well as compositions of components (`Composite`).

As the objectifier, the composite pattern is used by some other patterns, for instance, the *chain of responsibility* or the *command* pattern [42].

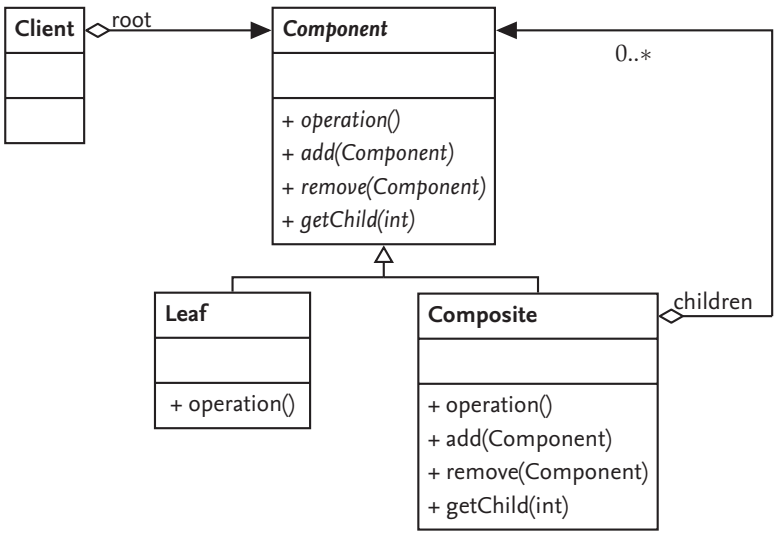


Figure 2.3: Class diagram of composite pattern [16, p. 164]

Design Pattern	GoF-Category	Uses Patterns	Description
Abstract Factory	Creational	Template Method	Creates families of related objects without specifying their concrete classes
Bridge	Structural	Adapter	Decouples abstraction from implementation so that both can vary independently
Chain of Resp.	Behavioral	Composite, Decorator	Decouples sender and receiver of request to let other objects handle it in between
Observer	Behavioral	Objectifier, Singleton	Defines a $1:n$-dependency between objects to notify all objects that depend on the changing of the subject's state
Strategy	Behavioral	Objectifier	Defines a family of algorithms while individually encapsulating each one to make them interchangeable
Visitor	Behavioral	Objectifier	Separates an algorithm from an object structure on which it performed

Table 2.2: Design patterns for typical software problems relevant for this work [16, 42]

Design Patterns for Typical Software Problems

In the following, we exemplary describe some of the design patterns for typical software problems, their intents and their relationships to other patterns. In Table 2.2, we list a subset of these patterns, identified by Zimmer [42], which are based upon the basic design patterns and are relevant for this work.

Abstract Factory. As an object creational pattern, the *abstract factory* can be applied to abstract object instantiation and lets the developer avoid depending on concrete classes [16, p. 87 ff.]. The goal of an abstract factory is to abstract the creation of families of related or dependent objects, which we call *products* in the context of this pattern. To this end, we introduce an abstract factory class, which offers methods to create different families of products. For each product family, we now introduce a concrete factory class that implements the factory methods of the abstract factory. We now depend solely on which concrete factory class we get and let it handle product creation instead of instantiating products ourselves.

Using the factory pattern, developers only have to concentrate on *when* to create an object, instead of concerning *which one* and *how* to create it. Such an abstraction contributes to maintainability and extensibility of the system. Abstract factories are often implemented using *factory methods*, using template methods that create objects [42]. Because of its specific intent of abstracting the creation of families of products and using another pattern within, the abstract factory is categorized as a pattern for typical software problems (cf. Table 2.2).

Strategy. As mentioned above, the *strategy* is a more concrete version of the objectifier pattern [16, p. 315 ff.]. Structurally, both patterns look the same (cf. Figure 2.2). However, they differ in their respective intention. The objectifier is used to objectify any behavior whereas the strategy is applied specifically to objectify similar yet different, interchangeable algorithms.

Observer. A more complex behavioral object pattern for typical software problems is the *observer* pattern [16, p. 293 ff.]. Partitioning a system leads to problems: either the parts of the system are tightly coupled and not so reusable, or a loosely coupled interaction has to be created to maintain reusability. The observer addresses this issue. It is applied to notify dependent objects automatically, if the object they depend on changes its state. To this end, a one-to-many relationship between the notifying object (the so-called *subject*) and the dependent objects (*observer*) is established.

We illustrate this pattern in Figure 2.4. In the abstract class `Subject` we provide different operations to handle the observers, which have to be attached, detached and, of course, notified. The observers have to implement an `update` operation, which is called by the `notify` method of the `Subject`. The `ConcreteSubject` can provide additional operations, which can be used within the update method, specified by a `ConcreteObserver`, e.g., operations to learn the subject's state. In this example, the states of the subject and the observers are simply synchronized.

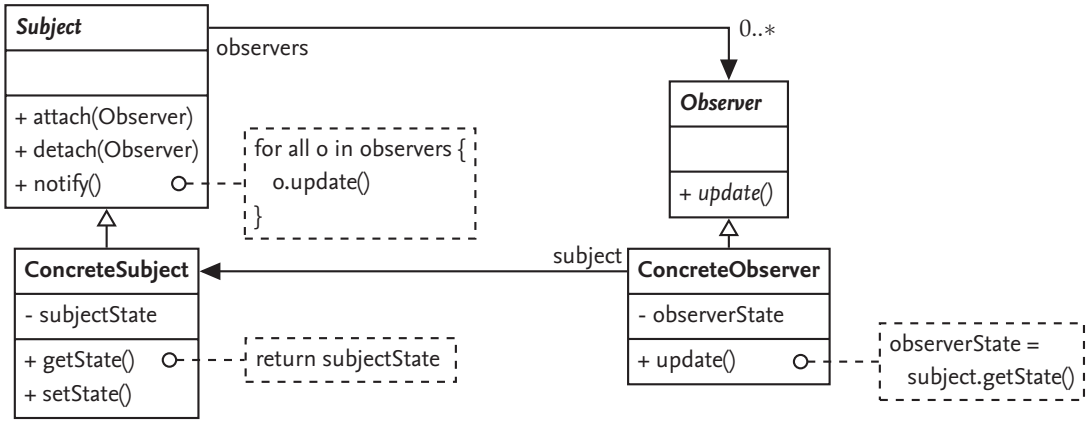


Figure 2.4: Class diagram of observer pattern [16, p. 294]

Using the Observer pattern, different objects that depend on the subject's state can be updated on each change of the subject's state. Because different observers simply attach and detach themselves, the subject does not have to make assumptions about which objects have to be notified. This offers a loosely coupled way of interaction between dependent objects leading to a system where subjects and observers can be reused independently.

Additional Design Patterns

In Addition to the patterns documented by Gamma et al. [16] and Zimmer [42], more patterns have been documented in recent years, such as the *extension objects* or *role object* patterns [15, 8].

Extension Objects. The *extension objects* pattern aims at anticipating that an object's interface needs to be extended in the future. Therefore, extension objects can be defined, adding additional interfaces to an object [15]. To this end, objects might introduce a `getExtension` operation that is used to query whether that object currently has a particular extension. Extensions need to implement an `Extension` interface, and need to be registered with a reference to the object they extend. To identify a particular extension within the extensible object, the `getExtension` operation can, for instance, be parameterized with a string that is used for extension identification. Using this pattern, we can register various extension objects with an extensible object, query the object for a particular extension and use the interface of the extension to perform new operations on the object.

Role Object. The role object pattern is the realization of the role modeling paradigm in object-oriented languages (cf. Section 2.3). When using objects, we might need to access different roles of this object. For instance, if we handle persons in our software, and some of these persons are customers, while others are employees, we cannot transparently access these roles, instead, we would have to either model customer and employee-specific behavior into our person representation, or we could use subclassing to differentiate between customers and employees. However, if one of our employees is also our customer, we would need two representations for this person's two roles. The *role object* aims to solve this particular problem by objectifying role-specific behavior into single role objects (i.e. customer-role and employee-role), and attaching these roles transparently to our abstraction (i.e., person). This is realized, similar to the extension objects pattern, by creating a composite structure of objects with core and role components, and offering the possibility of adding and retrieving various roles to core components. This way, we can transparently treat a person as a customer or an employee (or both) by simply attaching the respective role components to the person and retrieving the required roles when needed.

Extension objects and *role object* are quite similar patterns, both objectifying behavior and dynamically attaching various extension or role objects to their subject, respectively. This way, both patterns mainly contribute to the extensibility of the system.

2.2 Software Product Lines

For a long time, software development focused mainly on developing one single product that fits the needs of all customers, a so-called *standard software* that can be purchased *off the shelf* such as Microsoft Word [2]. While this sort of standard software is suitable for many domains, however, there are domains where the configuration of software is desirable and different variants of this software are required. To develop such variant-rich software systems, in recent years, the concept of *software product lines* (SPL) gained momentum as a reaction to the oncoming needs for mass customization in the context of mass production.

The general idea of SPLs is to enable developers to tailor their software products to the individual requirements of their customers [2, 11, 27]. Compared to individual software development, in SPL development also the variable parts of applications are modeled and reused. While other approaches such as frameworks also allow individual customization, this reuse of variability is the unique characteristic that sets SPL development apart from other approaches. This makes SPL development balance between allowing customization and reducing cost.

In other domains, for example the production of cars, it is common nowadays to be able to choose from highly customizable products. The same kind of customization has emerged in the context of software systems. But not only should the end customer be able to configure the product according to his wishes, but also could SPLs be part of a larger end product such as printers containing a firmware that is developed as an SPL. The Software Engineering Institute defines SPLs as follows [11]:

> *"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."*

Hence, an SPL encompasses a number of similar, yet distinguishable software products that are developed for a specific domain. Moreover, SPLs comprise features, which are increments in prod-

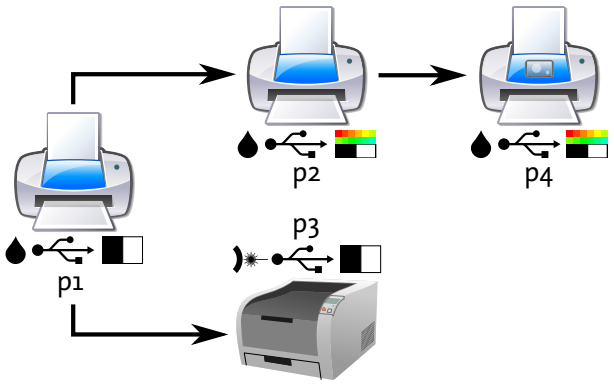


Figure 2.5: Printer example SPL

uct functionality visible to any stakeholder [2]. However, the difference to standard software is that products of an SPL share a common set of features, from which some are core assets comprising the core functionality required for the SPL's domain. Benefits of software product lines include tailor-made software instead of *"one-size-fits-all"* standard software, reduced development costs, improved product quality and a lower time to market [2, 11].

As an example to describe the concepts of software product line engineering, we take an exemplary look at firmware development for printers. Printers are a product family for a very specific domain, where the products have quite similar, yet slightly different, features and a lot of basic functionality shared by the different models. This example is based on the fact that Hewlett Packard, for instance, develops its firmwares for printers, scanners, copiers and fax devices as a software product line [2].

In Figure 2.5 we illustrate our printer example with four products, *p1* to *p4*. The first product, *p1*, offers a basic black & white inkjet printer with USB connectivity. Based on that, printer *p2* has been developed and extended with color printing. Printer *p3* greatly differs from *p1* by being a laser printer instead of an inkjet. However, both use the same USB connectivity and both might use the same image processing because of both being black & white printers. Printer *p4* on the other hand is based on *p2* and extends it by offering a display for configuration purposes.

What we can see in this example is that all the printer variants share the same set of features consisting of *{Inkjet, Laser, Black&White, Color, USB, Display}*. Some of these features, such as *USB*, are core assets, which all the variants are based upon. Other features, for instance, *Black&White* or *Color*, are also core assets because they provide general functionality required to create a printer. Other features, such as the display, might extend the product by optional functionality. Thus, our product line consists of similar yet different products sharing a single, common set of features. Since some features are used in multiple variants of our product line, for instance, the *Color* feature in *p2* and *p4*, we can see that product lines aim at exploiting these commonalities and variabilities by reusing features for different variants, implementing a functionality in one single feature for all products that require it.

In order to identify commonalities and differences of products in a particular domain to implement the functionality in reusable artifacts, and finally to develop single products, the process of SPL engineering is divided into *domain engineering* and *application engineering* [27].

[2] Hewlett Packard in the SPLC Product Line Hall of Fame: http://splc.net/fame/hp.html

Domain Engineering. The domain engineering of a software product line consists of identifying which products with which functionalities are required for that particular domain. Moreover, the commonalities and differences of these products must be determined, on which basis the features can be defined. To model the variability, i.e., commonalities and differences within a software product line, *feature models* have been introduced by Kang et al. [21], which we introduce in Section 2.2.1. The variability identification and modeling of the product line is described as the *problem space* of the SPL engineering [12]. Furthermore, the domain engineering comprises the implementation of reusable implementation artifacts, which is described as the *solution space* of the SPL engineering [12].

Application Engineering. The application engineering of a software product line aims at creating single variants from the product line, based on the variability model and implementation artifacts of the domain engineering [27]. To this end, a set of features is selected, which represents the desired variant, and the features are combined, leading to a single, customer-tailored product.

In the following sections, we describe the aforementioned feature models as a way of modeling variability in SPLs. Moreover, we give a detailed overview of *feature-oriented programming (FOP)*, a recent implementation approach for SPLs supporting clear modularity of features. Furthermore, we describe feature interaction, which is a major problem faced in product line engineering.

2.2.1 Feature Models

A crucial phase in software product line engineering is the domain engineering, where the domain is analyzed regarding the required products and their commonalities and differences, which are then used to identify the necessary variability of the product line. This variability encompasses the features and their relationships. To model this variability, a *feature model*, introduced by Kang et al. [21], can be used. Feature models are hierarchical tree structures that describe the features and their relationships. A graphical representation of feature models is called *feature diagram*. In Figure 2.6, we depict a feature model for the previously introduced printer example SPL.

There are different relations, also called *variation types*, between parent features and their subfeatures and in between a group of subfeatures that can be described using feature models [21, 12]. Subfeatures can only be included in the variant if the parent feature is included. Features can be characterized as *mandatory* or *optional*. Mandatory features have to be included in every variant (cf. feature *USB*), while optional features may be included or not (cf. feature *Display*). Features within a feature group are called *siblings*. There are different constraints, describing how siblings can be related to each other.

And

Every mandatory subfeature of that group has to be included, every optional subfeature may be included (cf. subfeatures of *Printer*).

Or

One or more subfeatures of that group have to be included (cf. subfeatures of *Color*).

Alternative

All subfeatures of that group are mutually exclusive and thus exactly one of them has to be included in each variant containing the parent feature (cf. subfeatures of *Technology*).

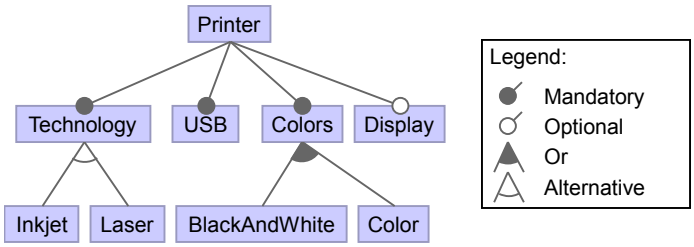


Figure 2.6: Feature model for printer example SPL

Furthermore, cross-tree constraints may be specified, expressing additional relations between features that cannot be expressed within the feature model itself [7]. A common way to express such cross-tree constraints are propositional formulas. For example, a possible constraint in our printer SPL could be that laser printers require a display, which would be expressed as follows:

$$Laser \Rightarrow Display$$

Feature models express all possible and valid combinations of features, hence, possible variants. A selection of features from the feature model is called a *configuration*, whereas a configuration is *valid* if it is consistent with all constraints in the feature model.

2.2.2 Feature-Oriented Programming

In order to develop software product lines efficiently, variability mechanisms connecting the problem and solution space are necessary, projecting the variability of the problem space onto code level. Common techniques include, for example, annotative approaches, such as preprocessors (e.g. C/C++), which are used to annotate code fragments with their corresponding features. Selecting a set of features leads to a selection of respective code fragments, thus, realizing a variable software system.

However, surrounding feature code with annotations leads to highly unreadable, tangled code that is excessively difficult to develop and maintain. Thus, the idea of separating features completely emerged. To this end, compositional approaches to implement variable software systems have been introduced, such as *feature-oriented programming (FOP)* [28, 6]. The core idea of FOP is to decompose a program into a set of features. In application engineering, we can create a configuration by selecting a subset of features. Moreover, we determine the order in which the features are applied. With this configuration, we can trigger a generation process that composes only the selected features and outputs the desired variant of the SPL.

Different approaches and languages exist to implement feature-oriented SPLs such as AHEAD [6] or FeatureHouse [5]. The core idea of these approaches is to enable the developer to define software artifacts modularly and give them possibilities to *extend* these artifacts in subsequent features. This extension of artifacts in subsequent features is called a *refinement*. To this end, the code for each feature is encapsulated in one *feature module*. Even though they are a new paradigm to product line implementation, feature-oriented approaches are still based on standard programming languages such as the object-oriented programming language Java, which we focus on in this work. Using Java in the context of feature-oriented programming, we get the possibility to refine classes via features. This means, multiple features can contribute to the same class. While the definition of a class contribution within a feature is commonly referred to as a *role* [39], to avoid terminology

conflicts with concepts of the role modeling paradigm (cf. Section 2.3), we call this a *feature-oriented role*.

In order to manufacture a single variant of the product line, the features and their corresponding feature modules have to be composed. The approaches of composing feature-oriented SPLs mainly differ in their composition mechanism. While FeatureHouse applies the concept of *superimposition* to compose artifacts [5], AHEAD additionally supports *mixin-based inheritance*. Composing classes via mixins leads to an inheritance tree, where each refinement creates a new class that extends the refined class [6, 39]. In this work, we concentrate on FeatureHouse as the most advanced feature-oriented implementation approach and, thus, on superimposition as compositional approach.

Using an object-oriented base language like Java, refinements offer the possibility to add or extend classes. Classes can be extended by adding new methods or fields or changing existing ones. Methods can be overwritten or extended by adding code and using the FeatureHouse-specified keyword `original` to refer to the original code of the refined method. In Figure 2.7, we illustrate a toy example for our printer example SPL (cf. Figure 2.5 & Figure 2.6) using FeatureHouse.

FeatureHouse relies on the general concept of *feature structure trees (FST)*. FSTs represent any kind of software artifact with a hierarchical structure and describe the modular structure of an artifact while abstracting from language-specific details. Every artifact of a language that has a hierarchical structure can be expressed with an FST. Java, as an object-oriented programming language, offers a hierarchical structure and, thus, can be expressed in FSTs. A Java class structurally consists of the following layers: a package declaration where each identifier expresses one layer, a class declaration as well as field and method definitions. Transforming such a class into an FST, the root package is used as the root node, while the subpackages follow as child nodes, recursively. The class declaration acts as the child node of the last subpackage. The fields and methods of the class are now the child nodes of said class and the leafs of the FST. In Figure 2.8, we illustrate an FST for the FeatureHouse-code of our printer example SPL that we depict in Figure 2.7.

The basic idea of FeatureHouse is not to simply compose the software artifacts, but to superimpose their respective FSTs, resulting in a language-independent composition mechanism. The result of the composition is the desired variant as described in the valid configuration of features. To determine, which artifacts to superimpose, FeatureHouse uses the information provided by the FST, consisting of names, types, and the relative positions of elements. Packages, classes, fields and methods, having the same name and also the same parent node, are superimposed, respectively. Method bodies are either overridden if the keyword `original` is not used, or composed by placing a call to the original code of the refined method at the position of the `original` keyword of the refining method.

For our printer example, such a composition would look like the right hand side of Figure 2.7, which is the implementation of the composed FST in Figure 2.8d. Here, we depict the composed classes, assuming a configuration where all three features of the left hand side are selected. Feature *Inkjet* would create a class `InkjetPrinter` (cf. Figure 2.8a) and a class `Ink`. In feature *USB*, because the class `InkjetPrinter` has been defined before, it is now refined by adding the method `receive` (cf. Figure 2.8b), while also the class `Usb` is added. The feature *Black&White* now refines the `print` method of class `InkjetPrinter` (cf. Figure 2.8c), introduced in feature *Inkjet*, by calling the original implementation of the method in Line 28 in Figure 2.7 and adding an additional statement.

Feature *Inkjet*

```
1  package printer;
2  class InkjetPrinter {
3    Document doc;
4    void print() {
5      System.out.println(doc);
6    }
7  }
8
9  package util;
10 class Ink {
11   void updateFill() { /* ... */ }
12 }
```

Feature *USB*

```
13 package printer;
14 class InkjetPrinter {
15   void receive(Document doc) {
16     doc = Usb.getDoc();
17   }
18 }
19
20 package util;
21 class Usb {
22   Document getDoc() { /* ... */ }
23 }
```

Feature *Black&White*

```
24 package printer;
25 class InkjetPrinter {
26   Ink blackInk;
27   void print() {
28     original();
29     blackInk.updateFill();
30   }
31 }
```

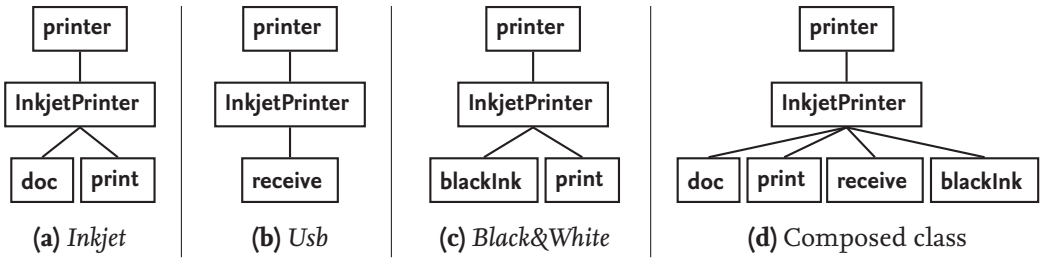
Composed Classes

```
32 package printer;
33 class InkjetPrinter {
34   Document doc;
35   Ink blackInk;
36   Ink colorInk;
37
38   void print() {
39     System.out.println(doc);
40     blackInk.updateFill();
41   }
42
43   void receive(Document doc) {
44     doc = Usb.getDoc();
45   }
46 }
47
48 package util;
49 class Ink {
50   void updateFill() {
51     /* ... */
52   }
53 }
54
55 package util;
56 class Usb {
57   static Document getDoc() {
58     /* ... */
59   }
60 }
```

Figure 2.7: Toy example for feature-oriented implementation of printer example SPL (cf. Figure 2.5 & Figure 2.6) using FeatureHouse with Java

2.2.3 Feature Interactions

One recurring problem in FOP, but also in software product lines in general, are *feature interactions* [6]. Generally, features are perceived as being isolated. However, collaborations of features are necessary in order to implement complex software systems. Feature interactions describe every kind of conflicting or collaborative implementations of feature modules, caused by a desired variability in the feature model. On the one hand, the implementation of a functionality may not always depend solely on one feature. Two or more features have to collaborate for implementing the functionality, as there will be code that is only necessary if the interacting features are included.



(a) *Inkjet* **(b)** *Usb* **(c)** *Black&White* **(d)** Composed class

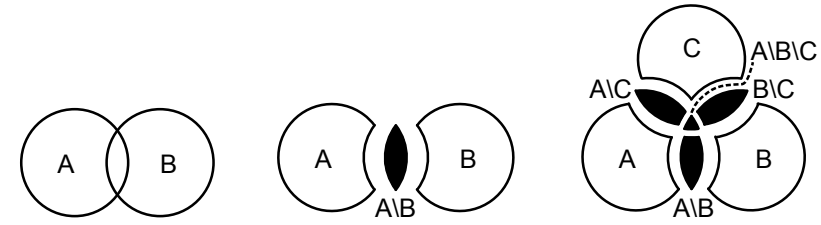
Figure 2.8: FeatureHouse FSTs of feature-oriented implementation of class `InkjetPrinter` of printer example SPL (cf. Figure 2.7)

On the other hand, unwanted interferences between two or more features may exist that may lead to unpredictable program behavior. This could occur, for instance, when two or more features modify the same resource, such as a global variable.

A special case of feature interaction is the *feature-optionality problem* (or *optional feature problem*) [23]. When the implementation of a functionality depends on two or more optional features, it may overlap. While including those features separately is straightforward, including two or more overlapping features always leads to additional code that has to be included if and only if a respective combination of these overlapping features is included. The problem lies within the nature of modularity: within feature modules, we have no knowledge, which other features are included. If they are optional, we cannot make any assumptions about other features. Thus, we cannot implement the additional code that is required for overlapping features within their corresponding feature modules. As a result, regardless of a variant's validity, it may not be producible due to implementation issues.

As an example, consider our printer SPL from Figure 2.6 again. Imagine the optional display should be used to configure any kind of configurable subsystems of the printer. Now imagine, we would expand our product line by the completely independent feature *Wi-Fi*, which would introduce Wi-Fi connectivity. This Wi-Fi connection should of course be configurable using the display, but that display is optional, too. Thus, we would have two completely independent features, which can each be included independently without any concerns. However, if both features are included at the same time, they would be forced to interact. Moreover, within these features we would have no knowledge of the other feature being present or not. Therefore, we could not implement the functionality of the Wi-Fi being configurable through the display in any of these features.

Derivative Modules. *Derivative modules* (or *derivatives*) are one possible solution to the feature-optionality problem [26, 23]. In general, for every combination of optional features, which exhibit interacting implementations of a functionality, the respective, additionally required code is extracted into a new feature module, which is included if and only if all of the interacting features are included. Illustrating this process, in Figure 2.9a, we can see features A and B coinciding, leading to overlapping and thus interacting parts of code. We illustrate a possible solution in Figure 2.9b, where the additional parts of code are encapsulated in a new feature module A\B, called *derivative module*. In Figure 2.9c, we show the solution of three overlapping features. For every combination of features, where interactions occur, the corresponding code is modularized within a derivative module.



(a) Features A and B **(b)** Derivative A\B **(c)** Derivative A\B\C

Figure 2.9: Illustration of derivative modules [23]

Considering the earlier example of provoking the feature-optionality problem by introducing a configurable Wi-Fi feature, we could solve this by encapsulating the interacting parts within a derivative module. Thus, we would introduce the basic functionality of the display in feature *Display*, whereas the basic functionality of feature Wi-Fi would be introduced in feature *Wi-Fi*. Thus, both features would be usable independently without the other one. Now, to make the Wi-Fi configurable through the display, we would introduce a new derivative module *Display\Wi-Fi*, which encapsulates exactly the functionality necessary to fulfill the interaction, extending the display with configuration possibilities for the Wi-Fi. This feature shall only be included if and only if both interacting features, *Display* and *Wi-Fi*, are included in the configuration. To this end, we could formulate a constraint in the feature model as follows:

$$Display \wedge Wi\text{-}Fi \Leftrightarrow Display \backslash Wi\text{-}Fi$$

As described, derivative modules solve the feature-optionality problem by creating a new feature module, encapsulating the conflicting or collaborating parts of code. Using this solution, we neither have to change the behavior of our program nor do we reduce its variability, since we are able to implement all the features of our feature model. The major drawback of derivative features is the increased effort that developers have to make. Also, the code is not modularized perfectly, as, in our example, parts of the display functionality would be introduced within the derivative *Display\Wi-Fi* instead of the *Display* feature itself. Moreover, the number of modules increases quickly. Still, derivative modules provide a strict separation of concerns, leading to the desired variant and, thus, are a suitable solution for the feature-optionality problem.

2.3 Role Modeling

As we briefly mentioned in Section 2.2.2, we concentrate on object-oriented programming (OOP) languages within this work. In OOP, developers are mostly concerned with classes and objects as well as the relationships between them, i.e., object composition via references as well as class inheritance. However, modeling an object-oriented design using classes and objects leads to a concrete design decision and, thus, to a definite implementation. When wanting to consider the collaborations of objects, we notice that an object can play multiple roles depending on our view on the object and which collaboration we consider. However, when concentrating on classes and objects only, we cannot capture the different roles objects play in different collaborations.

To this end, the concept of *role modeling* emerged and was first explicitly used by Reenskaug et al. [30]. Being a general modeling concept, role modeling can be applied to OOP, as well as other

modeling approaches, such as feature models. Role modeling bridges the gap between a design idea (i.e., collaborations that have to exist) and a concrete design (e.g., a class diagram). While we focus on giving a general description of role modeling, we use the term *object* to describe elements of definite design, which can be objects of OOP but also, for instance, features of feature models.

Roles offer a new kind of view on such objects, focusing on dynamic collaboration and interaction instead of a static structure [33]. A *role type* describes the view an object has of another. An object can be assigned multiple role types, which means that the object *plays* these roles. Because of an object acting according to various roles, different clients may perceive different views of the same object. Moreover, a role type can be assigned to multiple objects. Hence, roles describe a part of an interaction that an object is involved in, in a specific context, which is captured by the *role model*. Usually, roles and role models are not first-class programming language constructs, but higher-level design concepts, which means that roles can be mapped to object-oriented language constructs such as classes and objects [33]. However, with *Object Teams*, Herrmann [19] developed a programming language featuring roles as modeling entities.

Benefits of using role modeling includes splitting an object, for instance, a class, into smaller pieces and, thus, being able to model different aspects of it while emphasizing the context-dependent parts. Moreover, role models abstract from definite design decisions and can be reused independently of definite design elements, such as classes or features. Because of that, role models can be easily composed by adding the roles of the other role model and introducing constraints and relations to existing roles (cf. Section 2.3.1). Nevertheless, role models are not definite. There can be various ways of modeling a collaboration, depending on the view that we actually want to capture.

In the following, we explain the notation of role models used in this work, and, afterwards present related work focusing on the application of role models to describe design patterns.

2.3.1 Role Model Notation

In contrast to classes and objects, roles capture different views on objects depending on their interaction with other objects. Consequently, class diagrams depicting a structure of classes and objects are not suitable to model roles. Since no unified specification or notation for role models exists, we base this work on the notation introduced by Riehle et al. [33], where roles are depicted as rectangles with rounded corners and different relations and constraints exist to model the interaction of roles and constraint the corresponding object relations:

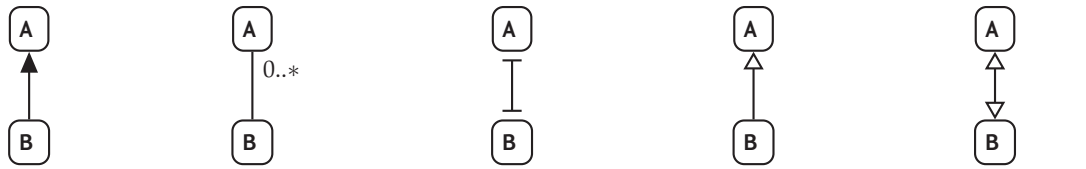
Use (cf. Figure 2.10a)
An object that plays role B uses an object playing role A.

Association (cf. Figure 2.10b)
An object that plays role A knows of an object playing role B, and vice versa.

Prohibition (cf. Figure 2.10c)
An object that plays role A must not play role B in the same context, and vice versa.

Implication (cf. Figure 2.10d)
An object that plays role B must also play role A.

Equivalence (cf. Figure 2.10e)
An object that plays role A must also play role B, and vice versa.



(a) Use **(b)** Association **(c)** Prohibition **(d)** Implication **(e)** Equivalence

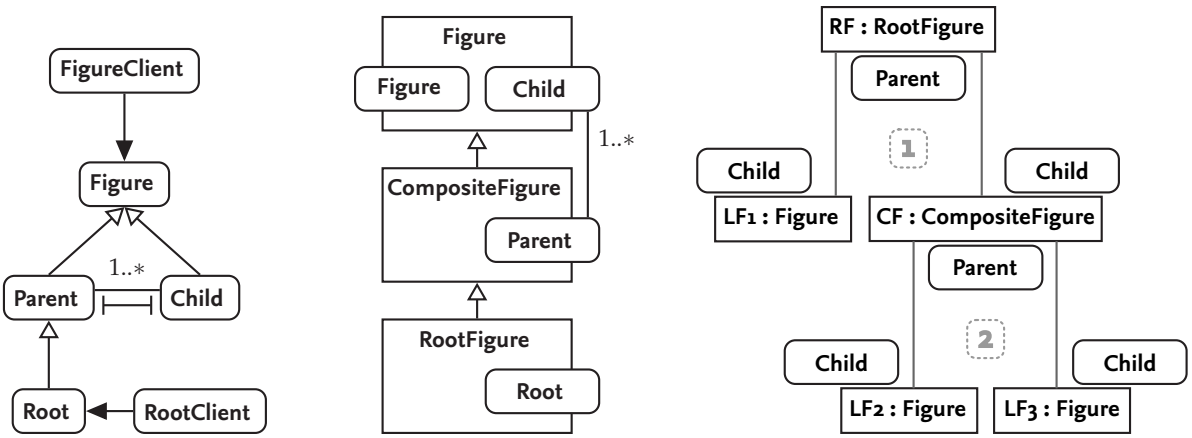
Figure 2.10: Role model relations [33]

We illustrate an example role model in Figure 2.11a that is adopted from Riehle et al. [33]. In this example, a hierarchical structure of figures is shown, where figure objects play roles to maintain a tree-like structure. Here we can see that an object playing the root role must also play the parent role. Objects playing the parent role comprise several objects playing the child roles. Objects playing the child or parent role also play the figure role. However, objects playing the parent role must never play the child role in the same context, and vice versa.

These roles can now be mapped to classes and objects to represent an implementation based on this role model. To this end, we can use a class ability diagram as we show in Figure 2.11b. Here we can see how the roles of Figure 2.11a are mapped to the classes `Figure`, `CompositeFigure` as well as `RootFigure`. The root role can, of course, only be played by objects of class `RootFigure`. Objects playing the root role also play the parent role, thus, the `RootFigure` is a subclass of the `CompositeFigure`, which plays the parent role. Because objects that play the parent role also play the figure role, `CompositeFigure` is a subclass of `Figure`. The class `Figure` can now also play the child role, which is maintained by a parent role as indicated with the role association between parent and child.

The role prohibition does not have any impact on the class structure, but constrains the roles objects play at runtime. In Figure 2.11c, we depict an object diagram for the figure hierarchy annotated with the corresponding parent and child roles. The role prohibition constrains that an object, playing a role, must not play the prohibited role *in the same context*. However, that does not constrain that an object cannot ever play both roles at the same time. We illustrate the meaning of a *context* in Figure 2.11c on the example of the object *CF* of type `CompositeFigure`. While playing the role child of *RF* in context 1, *CF* can also play the parent role at the same time, however, only in another context with different child objects. In this case, the role prohibition constrains that an object playing the parent role cannot reference itself as a child because then it would play both parent and child roles within the same context.

Using these notations, we can express role diagrams with basic and necessary constraints and afterwards implement our roles using a class ability diagram. Consequently, we are able to abstract our design from our implementation by considering collaborations and interactions of objects instead of a mere structure, which leads to a more detailed and reusable design that can be arbitrarily mapped to classes and objects while fulfilling the defined constraints [33]. However, the notation of role modeling is not fixed. There are different notation style that also differ in expressiveness, for instance, Reenskaug et al. [30] defines other constraints regarding different aspects of the collaborations. Moreover, the set of constraints can be extended by introducing new kinds of constraints for new domains.



(a) Role diagram **(b)** Class ability diagram **(c)** Object diagram with parent/child roles

Figure 2.11: Figure hierarchy example for role models from Riehle et al. [33]

2.3.2 Modeling Design Patterns using Role Modeling

As we expressed in Section 2.1, design patterns are general solutions to common, recurring design problems in object-oriented programming. Because of their nature of being a general description rather than be a definite design and implementation, describing design patterns using classes and objects cannot be suitable. In order to be able to model design patterns independently of their implementation, Riehle [31, 32] proposed modeling design patterns using role models.

Riehle [31] argues that role models, concerning the collaboration of objects instead of the definite class structure of a system, are quite suitable means to model design patterns. In Figure 2.12, we illustrate a role model for the observer pattern that we described in Section 2.1.3.

When we abstract from the example implementation of the observer pattern that consists of four collaborating classes, we are left with two roles, a *subject* and an *observer*. Since there is not much sense in the subject observing itself, we use a role prohibition to avoid this. An object that plays the subject role holds an arbitrary number of objects playing the observer role.

In Figure 2.13, we map the observer role model to the figure hierarchy of Figure 2.11. This mapping realizes the idea that parent figures listen to changes of their children. In Figure 2.13a, we illustrate the corresponding class ability diagram, where each figure plays the subject role and each composite figure listens to figures. Similar to the parent-child relation in Figure 2.11c, as we show in Figure 2.13b, an object of type `CompositeFigure` can play both roles, subject and observer, in different contexts, by playing both child and parent roles.

In role models, we do not need to care about implementation specific details such as interfaces or abstract classes, methods and fields. We completely concentrate on the collaboration we want to model. However, as we mentioned before, our role model actually depends on the view that we want to express. If we want to model the collaboration of methods, such as a notify method of a subject

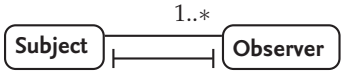
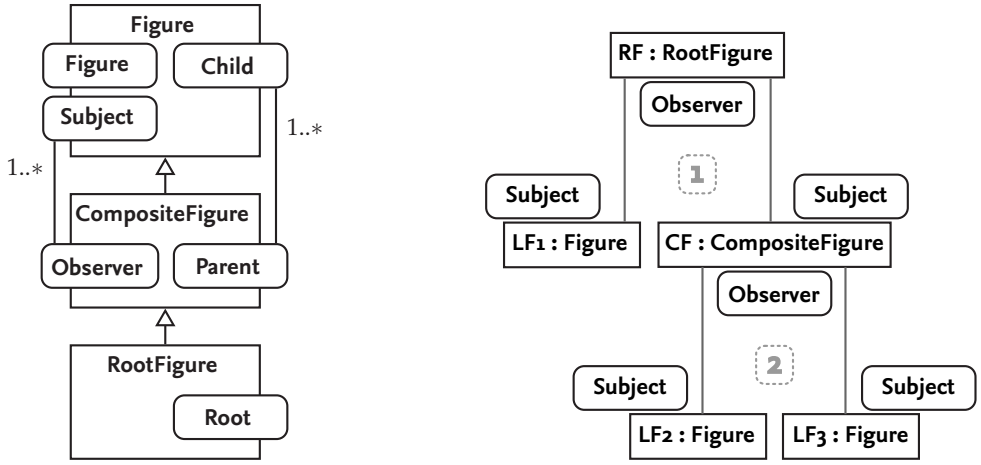


Figure 2.12: Role model example of observer pattern (cf. Section 2.1.3) [32]



(a) Class ability diagram **(b)** Object diagram with observer/subject roles

Figure 2.13: Mapping of observer role model to figure hierarchy example

calling its observer's update method, we are free to do so in our role model. Moreover, sometimes it is necessary to model more detailed constraints, such as the inheritance relation or an aggregation of two objects playing specific roles. To this end, we include further constraints to our role models in order to describe design patterns as detailed as possible.

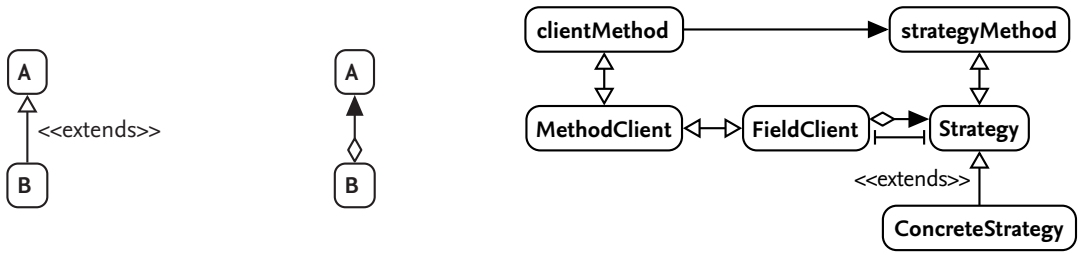
Since already others have determined defining inheritance constraints in role models in order to describe design patterns as useful[3], we adopt this constraint for this work. Moreover, we adopt the role aggregation constraint proposed by Riehle [32]. In Figure 2.14, we illustrate both these constraints:

Role implication with inheritance constraint (cf. Figure 2.14a)
An object that plays role B must inherit from an object playing role A.

Role aggregation (cf. Figure 2.14b)
An object that plays role B must hold an aggregation to an object playing role A.

In Figure 2.14c, we depict a more detailed role model for the strategy pattern. In this pattern, a `Client` holds a reference to a `Strategy`, and contains a `clientMethod` that calls the `strategyMethod` contained by the `Strategy` and implemented using multiple, interchangeable objects of type `ConcreteStrategy`. Because the `Client` plays two roles, holding the `Strategy` object and containing the `clientMethod`, we model it using two equivalent roles, *MethodClient* and *FieldClient*. The *MethodClient* contains the *clientMethod* while the *FieldClient* holds a reference to the *Strategy*, which contains the *strategyMethod* called (i.e., used) by the *clientMethod*. `Client` and `Strategy` must never be the same object, which is why we use a role prohibition to constrain this. The objects of type `ConcreteStrategy` must inherit from the `Strategy` in order to implement the `strategyMethod`. To this end, we use the *role implication with inheritance constraint* between *ConcreteStrategy* and *Strategy*.

[3] Prof. Dr. Uwe Aßmann* applies a *role implication with inheritance constraint* in his slides on *Design Patterns as Role Models* in the context of his lecture on *Design Patterns and Frameworks* – * Chair of Software Engineering, Technische Universität Dresden



(a) Role implication with inheritance constraint

(b) Role aggregation

(c) Role model for the strategy pattern

Figure 2.14: Detailed modeling of design patterns using role modeling

2.4 Metamodel-based Generative Software Development with EMF Ecore

Standard code-based development has been around for decades and is the foundation of software engineering. However, programming everything directly on source code level gets increasingly time-consuming and complex with growing software projects. Moreover, it is often very redundant such that similar fractions of code have to be written again and again.

The basic idea behind model-based development is to capture particular aspects of the software, such as structure, behavior or interaction, in domain models, which are abstract representations of that particular application domain [41]. The application is then implemented based on these models. Generative software development, on the other hand, focuses on automated source code creation based on more abstract, formal descriptions of the knowledge necessary to implement the software. Combining model-based with generative development, developers can not only orient themselves by domain models, but also exploit these models for code generation purposes [41]. We contrast the workflow of model-based and generative development in Figure 2.15. In mode-based development, we only use the model for orientation, but manually write the code. In contrast, in generative programming, we partially generate code in addition to manually written code.

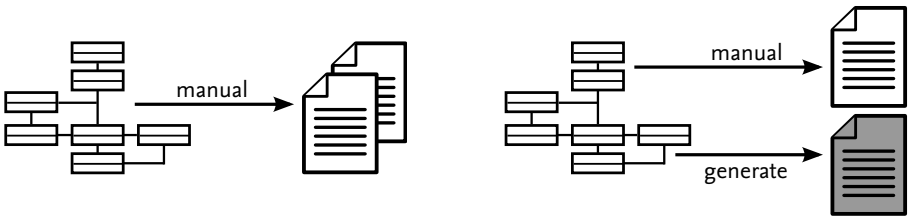
A metamodel is a concept used to specify the application domain. While a model abstracts one particular application, a metamodel describes what elements the model can consist of and their possible relations, thus, a metamodel is a *model of a model*. A model that conforms to its metamodel is called an *instance* of that metamodel. The *Meta-Object Facility (MOF)* is a standard of the *Object Management Group (OMG)* for model-driven engineering providing a metamodeling architecture consisting of four modeling layers:

Meta-meta model (M3)

The meta-meta model at the top layer is called the M3 layer and conforms to itself, thus, meta-meta models are described using their own elements. A meta-meta model is the language used by MOF to build metamodels.

Metamodel (M2)

A metamodel, which is the M2 layer, conforms to its meta-meta model. A prominent example for such a metamodel is UML 2.0 metamodel, which is the metamodel that describes UML models.



(a) Model-based development **(b)** Generative development

Figure 2.15: Comparison of model-based and generative development

Model (M1)

The model, or M1 layer, conforms to its metamodel and is used to describe logical or physical data or processes, such as UML class or sequence diagrams.

Data (M0)

The M0 layer, or data layer, is used to describe real-world objects, for example, an instance of a class diagram.

The *Eclipse Modeling Framework (EMF)*[4] is described as a modeling framework and code generation facility, creating a structured data model that is open to be used by other tools. *Ecore* is the metamodeling notation of EMF, which is based on *Essential MOF (EMOF)*, a variation of MOF consisting only of essential parts. Using an Ecore metamodel to describe our model elements, we can generate corresponding Java code for this model using the code generation facility. EMF includes various support facilities to manage the model, for instance, the automated resolving of containment hierarchies (explained below), which makes it easy to traverse the model in both directions of the hierarchy. Benefits of creating a metamodel, based an a widely used metamodeling approach such as Ecore, is that we gain the possibility of applying plenty of general tools, which have been developed for EMF models, to our model.

The first step of creating a model with EMF is to create an Ecore metamodel describing the elements of our structured data model. In Figure 2.16, we illustrate an exemplary Ecore metamodel of a library. First level entities in Ecore are `EPackages`, which are not illustrated in this example. Within these packages, we can model *EClasses* such as `Library`, containing EAttributes such as `name` of type `EString`. An `EClass` can extend multiple other `EClasses` via inheritance. Moreover, an `EClass` can hold references to instances of another `EClass` with a specified cardinality, such as `Library` holds various instances of `Book`. Such references have cardinalities and can optionally be defined as containment hierarchies, which means that, in this case, a `Book` is physically contained by a `Library`. This containment hierarchy is automatically managed by EMF and can be traversed in both directions (i.e., `Book` holds a parent reference to its `Library`). An object can only be contained in exactly one containment reference, thus, containment references build up a tree structure of elements. Moreover, we can optionally define *opposite* relations, such as the *library* relation of `Book` is the opposite of the *books* relation of `Library`. This opposite relation is also managed automatically. If we insert a `Book` into a `Library`, the *library*-reference, held by the `Book`, is automatically set.

We can now use this metamodel to generate code, so that we can employ the model in practice and work with it. In general, the Ecore metamodel is platform independent, which means, it is possible

[4] The *Eclipse Modeling Framework (EMF)*: http://www.eclipse.org/modeling/emf/

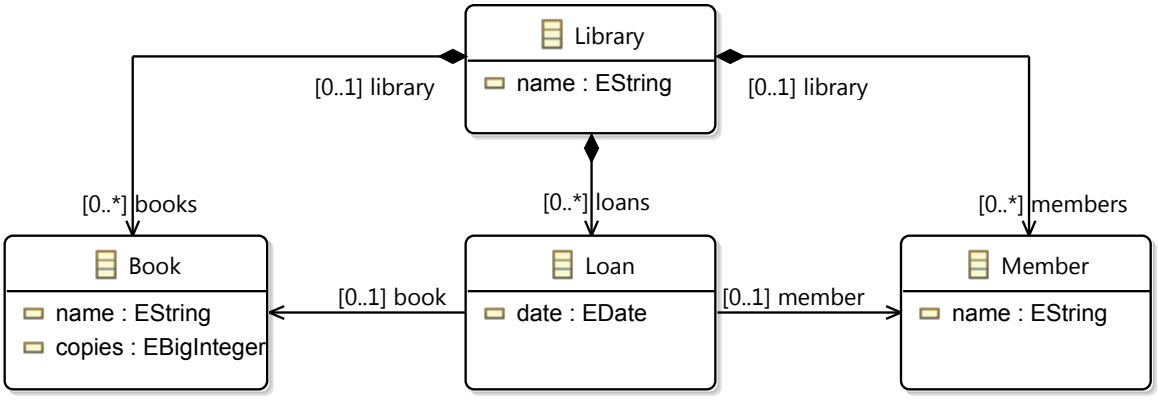


Figure 2.16: Example Ecore metamodel of a library

to generate code of any particular language to represent the model. However, EMF comes with a code generation facility for Java, so that we can generate Java code for our model out of the box. The mapping of Ecore to Java is straightforward. An `EPackage` is generated as a Java package. An `EClass` is transformed to a Java class, containing the attributes and references of the `EClass` as fields as well as getters and setters. The data types of attributes, such as `EString` or `EDate`, are automatically transformed to their corresponding Java types, such as `java.lang.String` or `java.util.Date`, respectively. EMF transforms the (multiple) inheritance relations of `EClasses` to a Java type chain using interfaces. The result is a complete Java representation of our metamodel that we can now instantiate to describe our application domains.

Various ways of instantiating and filling a structured data model exist, such as loading the data from datasets, using a graphical editor, or describing the data in a textual editor. For the library example above, a form with input fields to fill in books, members and loans would be suitable. For an AST of the Java programming language, for example, which we illustrate an extract of later in Section 4.2, a textual editor would be of best use. With EMF describing the data model and, thus, the abstract syntax of data or of a language, the necessary information to fill the data model, such as names and relations of elements, can be provided using a textual, concrete syntax with a textual editor. Different frameworks to develop programming languages and domain-specific languages using EMF exist, such as EMFText[5]. Using EMFText, we can enrich a metamodel with a textual syntax and automatically create an editor and a parser for the language using the corresponding EMF model as AST. JaMoPP, the Java model parser and printer, employs EMFText to describe the textual syntax for Java, offering an easily traversable Java AST and a matured parser for Java 1.5 syntax. We describe JaMoPP in Section 4.2.

2.5 Previous Work

In this section, we present our previous work concerning object-oriented design and design patterns in feature-oriented product lines. We started by reasoning about object-oriented design in general, and more specific, the applications of design patterns in FOP [36]. Our motivation was to take a closer look at what exactly the consequences of introducing class refinements to object-oriented languages were for the design of such software. We argued that design patterns, because of

[5] EMFText: http://www.emftext.org/

Feature *DirectedOnlyVertices*

```
1  class Graph {
2   void run(Vertex s) { /* ... */ }
3  }
```

Feature *TestProg*

```
4  class Main {
5   public static void main(String[] a) {
6    Graph g = new  Graph();
7    /* ... */
8    g.run(g.getVertices().next());
9   }
10 }
```

Feature *Cycle*

```
11 class Graph {
12  void run(Vertex s) {
13   System.out.println("Cycle?"
14    + CycleCheck());
15   original(s);
16  }
17  boolean CycleCheck() {
18   GraphSearch(new CycleWorkSpace());
19   return c.AnyCycles;
20  }
21 }
22
23 class CycleWorkSpace { /* ... */ }
```

Figure 2.17: *Feature template method* in *GPL* (simplified)

their modular fashion, would be suitable to support feature-oriented modularity within the solution space. This argument is based upon the fact that SPLs used to be implemented using design patterns to realize the variability. Our basic idea was to capture single concrete classes of patterns, such as a concrete strategy class, in single features, slicing design patterns by means of features. Moreover, while manually reviewing code, we identified a feature-oriented mechanism that was used quite frequently, which we called the *feature template method (FTM)*. Basically, an FTM resembles a template method pattern (cf. Section 2.1.3), but differs in using refinements instead of inheritance to define the concrete behavior of the hook methods. We depict an example implementation of the FTM that we identified in the *graph product line (GPL)* in Figure 2.17.

Here, we can see parts of the three features *DirectedOnlyVertices*, *TestProg* and *Cycle*. Within the feature *DirectedOnlyVertices*, we introduce a class `Graph`, containing an empty method `run` in Line 2. In the feature *TestProg*, we call this empty run method in Line 8. Now, in feature *Cycle*, we use class refinements to fill the run method of class `Graph` in Line 12. Because other features exist that also fill the run method, the original keyword is used. This way, the run method is filled by subsequent features with the algorithm that corresponds to the selected configuration.

However, design patterns capture best practices and well-known design experience. To support our hypothesis, there has to be evidence that patterns are actually applied the way we argued. To this end, we conducted a case study to prove that design patterns exist in FOP and that they can be applied across feature borders, thus, slicing the pattern by means of features [37]. We developed an initial approach for automated, family-based design pattern detection in FOP, using static analysis on the *abstract syntax tree (AST)* based on work of Heuzeroth et al. [20]. Moreover, we expanded our goals of the study to analyze whether design patterns are related to feature interactions and vice versa when they are implemented across features. Furthermore, we aimed at deriving guidelines on what aspects to consider when applying design patterns in FOP and how to apply them.

Our results have been quite promising. Using our detection approach, we were able to identify a number of design patterns. In Table 2.3, we list the patterns that we were able to detect in which product lines. However, we already identified deficiencies of our detection approach. Because of many false positives, we had to manually review the results, which was tedious work. Additionally,

Program Name	#Visitor	#Strategy	#Observer	#FTM
AHEAD	0 (0)	1 (32)	0 (0)	3 (3)
Berkeley DB	0 (0)	7 (23)	0 (0)	5 (5)
GameOfLife	0 (0)	1 (1)	1 (1)	1 (1)
GPL	1 (1)	2 (2)	0 (0)	18 (18)
GUIDSL	2 (2)	0 (16)	0 (0)	13 (13)
TankWar	0 (0)	0 (0)	0 (0)	1 (1)
Violet	0 (0)	7 (7)	0 (0)	6 (6)

X (Y) means that X of Y detected patterns are implemented across several features

Table 2.3: Detected feature-oriented design patterns in prior work [37]

our detection approach was applied to the AST that is generated by Fuji[6], which is inconvenient to use for family-based analyses. Consequently, our approach was only successful to some extent.

Nevertheless, we were able to detect instances of patterns that were implemented in a modular fashion across several features. In Figure 2.18, we depict the code of a decomposed visitor pattern that we detected in the GPL. We can see here that the base feature *DFS* introduces the *visitor* and the *element* of the pattern with their respective *visit* method in Line 2 and *accept* method in Line 9. On the right hand side, in feature *Number*, a concrete visitor `NumberWorkSpace` is implemented with its concrete implementation of the visit method in Line 27. What we also noticed was that this visitor pattern was not only decomposed but, in order to be implemented, combined with the exact FTM we illustrated in Figure 2.17. To this end, the same `run` method is introduced in Line 14 and refined by the *Number* feature in Line 35. We detected this combination of a design pattern and the FTM frequently, where the FTM is applied to introduce the concrete classes of the design pattern.

By encapsulating concrete classes of patterns within their one feature, we create a one-to-one mapping of features and concrete pattern classes. To enable this, we also have to model our pattern instance within the feature model. By including our design pattern and a definite implementation within our feature model, we blur the line between problem and solution space. We actually model a concrete design decision within the problem space of our application.

Because we were also concerned with structural feature interaction occurring by slicing design pattern implementations, we analyzed each detected pattern manually to qualitatively reason about the kind of feature interaction. We observed no direct interaction (i.e., no class refinements), because only new concrete classes (e.g., visitors) are added, but they are never refined afterwards. The only direct interaction using class refinements occurs within the FTM that is used to register and call the newly introduced concrete pattern classes. However, this way of implementing the patterns affects the variability because with optional visitors, all of the visitable elements have to be mandatory in order to avoid the feature-optionality problem (cf. Section 2.2.3).

Regarding guidelines for the use of design patterns in FOP, we noticed that decomposed patterns mostly occurred in sibling features. Hence, we argue that design patterns that are reflected within the feature model should be encapsulated within a feature group because it reflects the actual implementation and enhances understanding.

[6] Fuji – Extensible Compiler for FOP in Java: `http://fosd.net/fuji`

Feature *DFS*

```
1  public class WorkSpace {
2   public void preVisitAction(Vertex v) {
3    /* empty */
4   }
5   /*...*/
6  }
7
8  public class Vertex {
9   public void nodeSearch(WorkSpace w) {
10   /*...*/
11  }
12 }
```

Feature *DirectedOnlyVertices*

```
13 public class Graph {
14  public void run(Vertex s) {
15   /* empty */
16  }
17  public void GraphSearch(WorkSpace w) {
18   /*...*/
19  }
20 }
```

Feature *Number*

```
21 public class NumberWorkSpace
22    extends WorkSpace {
23  int vertexCounter;
24  public NumberWorkSpace() {
25   vertexCounter = 0;
26  }
27  public void preVisitAction(Vertex v) {
28   if (v.visited != true) {
29    v.VertexNumber = vertexCounter++;
30   }
31  }
32 }
33
34 public class Graph {
35  public void run(Vertex s) {
36   System.out.println("Number");
37   NumberVertices();
38   original(s);
39  }
40  public void NumberVertices() {
41   GraphSearch(new NumberWorkSpace());
42  }
43 }
```

Figure 2.18: Example of decomposed visitor implementation in *GPL* [37]

3 Variability-Aware Design Patterns

In this chapter, we introduce the idea of *variability-aware design patterns* as a means to exploit both variability concepts of FOP and object-oriented design patterns. First, we motivate the research on variability-aware design patterns in Section 3.1. Next, we explain our general approach, including the requirements and contributions in Section 3.2. Afterwards we introduce the concept of *family role models (FRM)* in Section 3.3. We explain the idea and contents of the variability-aware design pattern catalog in Section 3.4. Finally, we give a brief summary.

3.1 Motivation

Software design is a crucial task during software development. In *object-oriented programming (OOP)*, plenty of design principles emerged, resulting in the recurring use of design patterns in order to employ established and widely used solutions to common design problems (cf. Section 2.1) [16]. Since many design patterns focus on encapsulating variabilities, they have been applied to realize customization [2]. The main idea of *software product lines (SPL)* is to reuse commonalities and variabilities across the product space in order to increase customization and decrease cost [27]. To realize such customization, variability-awareness of realization artifacts is necessary. While object-oriented concepts offer modularity and variability, new implementation approaches for SPLs have emerged, annotative and compositional [2] as well as transformational [35], focusing on increasing variability-awareness for realization artifacts. However, even though most implementation approaches for SPLs are based on object-oriented languages, only little is known about the application and the impact of object-oriented design techniques in the context of SPLs.

Feature-oriented programming (FOP) [28, 6] is a compositional implementation technique for SPLs, for which most approaches, such as FeatureHouse [5], are based on object-oriented languages (cf. Section 2.2.2). FOP focuses on composing modules containing implementation artifacts that are decomposed along features, offering a new layer of design that results in the concept of *refinements*. As explained in detail in Section 2.2.2, refinements provide the possibility of incrementally adding functionalities to implementation artifacts, which leads to the possibility of implementing these artifacts scattered along several features.

Even though such SPL-specific implementation approaches are not required to implement an SPL, they facilitate the implementation of variability in realization artifacts. Using standard object-oriented languages, for instance, design patterns can be applied to realize variability and modularity in order to allow similar customization [2]. Many design patterns concern the decoupling and separation of variability. Due to this fact, we reasoned in [36] that using design patterns in SPL-specific implementation approaches, such as FOP, could be beneficial regarding the implementation of realization artifacts by exploiting the implementation of variability that design patterns offer. Moreover,

with features, FOP offers a new layer of design that has to be regarded when designing applications and, thus, when applying design patterns. Hence, in this work, we analyze the impact of design patterns on feature-oriented design on the one hand, and, on the other hand, how FOP affects the application of design patterns, while arguing that design patterns and a modular implementation approach, such as FOP, might coexist in symbiosis in order to introduce variability to realization artifacts.

To support this idea, we reasoned about the structure of such design patterns by proposing ideas on their variability-aware realization. This means, taking feature semantics and the possibility to decompose realization artifacts into account. We conducted a preliminary study on the application of design patterns in FOP by automatically detecting selected design pattern instances in selected case studies (cf. Section 2.5) [37]. We detected several design pattern implementations that are decomposed across several features, for instance, the decomposed instance of the visitor pattern in Figure 2.18. However, we could only give limited information on the general, variability-aware application of these patterns. Nevertheless, we learned that design patterns are applied in the context of FOP. Moreover, we analyzed the impact on feature interactions and learned that design patterns generally appear to rely on caller-callee relations.

Based on this, as we explained in Section 1.2, our goal is to reveal the variability-aware application of design patterns in FOP, thus, how exactly design patterns contribute to software product line design. To this end, we aim at creating a variability-aware design pattern catalog similar to the design pattern catalog of Gamma et al. [16]. Since design patterns are documentations of common solutions applied in real-world software design, we cannot just reason about the feature decompositions of design patterns, but need to collect evidence of their decomposition. To this end, we conduct a case study, consisting of automatically detecting design patterns in existing feature-oriented SPLs and analyzing the decomposition of the design patterns in terms of features. We explain our approach in Section 3.2.

In order to describe our findings of variability-aware design patterns in a general fashion disregarding definite design, we need a modeling approach that can be used to capture feature-oriented roles and their collaborations as well as the involved features with their relations. To this end, role modeling is perfectly suited. Hence, we introduce the concept of *family role models*, which describe the collaborations of feature-oriented roles, their containing features and the relations of these features in a general, language-independent fashion. We introduce family role models in Section 3.3. We use these family role models to describe the relations of feature-oriented roles and their containing features in order to create a catalog of general descriptions and application guidelines for variability-aware design patterns. In Section 3.4, we explain the ideas behind this catalog and its contents.

3.2 Approach

Because we want to capture the variability-aware implementation of design patterns, we need to conduct a family-based analysis. Only then, we can reliably and completely detect decomposed design pattern instances. In Figure 3.1, we outline the general approach of this family-based analysis.

First, we need to capture the collaborations of a design pattern and the general description of the pattern in order to automatically detect decomposed design patterns while regarding feature-oriented roles instead of the resulting class structure of a product. Role modeling allows capturing

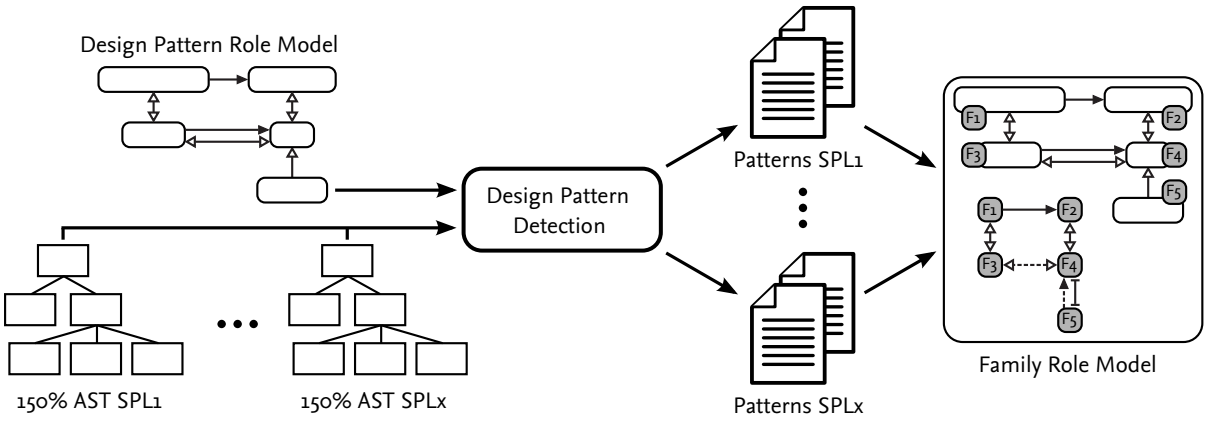


Figure 3.1: Overview of the general approach of this work

the structure of a design pattern independently of the concrete programming language they are used in by focusing on modeling collaborations (cf. Section 2.3.2). Hence, we can apply role modeling to capture the collaborating parts of design patterns and express constraints on their application, for example, that two roles (i.e., collaborations of a pattern) may not be played by two feature-oriented roles contributing to the same class.

To run an automated design pattern detection, we also need a system representation containing the variability information of the SPL. In our previous study, we employed Fuji[1], a fully-fledged compiler for feature-oriented Java code that conforms to the aforementioned FeatureHouse approach and is developed by extending the JastAddJ[2] compiler. With Fuji, we can compose a feature-oriented SPL disregarding the validity of the configuration. Thus, we can compose a maximal product containing all existing features. The result does not necessarily lead to valid Java files, however, Fuji annotates each element with its containing feature, thus, creating an AST containing all the information on the SPL when composing all features. However, Fuji only annotates each element with information on which feature *introduces* the element, but not which features *refine* it. When regarding references to other elements, for instance, method calls, we can only access the composed method with the annotation of its introducing feature, but not all existing declarations of this method with all its refining features. Hence, using Fuji it is not possible to reliably analyze collaborations across features and interaction between features, which means that deriving guidelines on the design pattern application from our results was only possible to some extent.

Because of this, we develop our own program representation based on the *Eclise Modeling Framework (EMF)*[3]. We especially regard reference resolving to other classes and elements such that we extend an existing reference resolver for Java code taking feature semantics into account. The result is an AST that contains all necessary variability information including resolved references to all existing declarations, leading to a system representation that can be used for family-based analyses. In reference to the notion of a 150% model with annotative variability realization mechanisms, we call this a *150% AST*.

Based on this AST, we develop a static, graph-based design pattern detection technique that takes

[1] Fuji – Extensible compiler for FOP in Java: http://fosd.net/fuji
[2] JastAddJ – Extensible Java compiler: http://jastadd.org/web/jastaddj/
[3] The *Eclipse Modeling Framework (EMF)*: http://www.eclipse.org/modeling/emf/

into account that, in FOP, feature-oriented roles instead of classes collaborate to build a design pattern. We conduct static analyses, such as structural program analysis, since dynamic analyses would require an executable piece of software. We develop our approach based on existing techniques for automated design pattern detection that have been developed for OOP, which we extend by feature semantics in order to create a family-based approach.

Using our approach for family-based design pattern detection, we conduct a case study on existing feature-oriented SPLs. In this case study, we try to detect instances of specific design patterns, which we can analyze afterwards regarding their application in the context of SPLs, thus, their decomposition along features. We first have to convert the feature-oriented Java code of an SPL to our 150% AST, the variability-aware system representation. On this 150% AST, we can run the pattern detection for each regarded design pattern. Afterwards, we manually review each result and derive guidelines and application rules for each design pattern leading to the variability-aware design pattern catalog.

In this work, we focus on specific design patterns that have been identified to be well-suited for variability implementation [2]. These design patterns all focus on encapsulating and decoupling varying behavior and, thus, to realize modularity, which is why we focus on detecting decomposed instances of these patterns. Moreover, because of their nature, by encapsulating specific parts and modularizing a system, these patterns all resemble fairly specific class and objects structures that facilitate the pattern detection.

3.3 Family Role Models

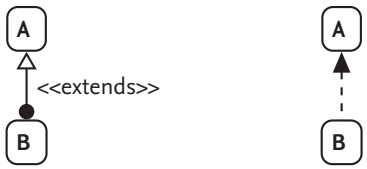
Design patterns are usually described using class diagrams, however, as we explained in Section 2.3.1, role modeling is well suited to describe the general collaborations of a design pattern instead of a definite design. We need to extend the notion of role modeling in order to capture the following information necessary to describe design patterns in the context of SPLs:

Collaborating roles of a design pattern
: Similar to regular role models used for describing design patterns, we need to capture the roles of a design pattern. These roles are played by feature-oriented roles as well as their method declarations.

Mapping of roles to features
: Additionally, in order to capture the feature distribution of a design pattern, we need to capture which role is played by which feature.

Collaboration of participating features
: To derive guidelines and application rules for SPLs, we need to capture the relations of the participating features in terms of the feature model within the role model.

We introduce *family role models (FRM)* as a concept for describing variability-aware design patterns by combining two role models, one for the collaborating roles of the design pattern and the other one for the collaborations of features. By describing features and their relations with role modeling, we can concentrate on the general constraints between these features that have to be met instead of describing a specific feature model. According to Alves et al. [1], certain feature relations can be expressed using different, yet semantically equal feature models. Thus, using role modeling instead



(a) Multiple child roles **(b)** Requires

Figure 3.2: Notation for family role models

of a feature model to describe these feature relations allows us to disregard these varieties and to cover different feature models expressing semantically equal feature relations at once.

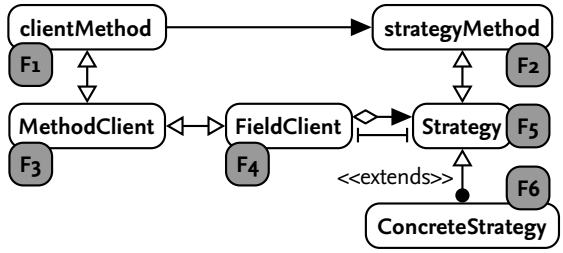
Regarding the role model concerning collaborating roles of design patterns, we use the modeling notation as we introduced in Section 2.3.1 including the extensions for describing patterns in a detailed fashion that we introduced in Section 2.3.2. Moreover, we extend the *role implication with inheritance constraint* by the possibility to express that there have to be multiple different objects playing the child role, which could not be expressed before (cf. Figure 2.14c). This constraint is necessary to express variabilities in design patterns realized using inheritance. In Figure 3.2a, we depict the notation for this constraint that extends the *role implication with inheritance constraint* by the • that implies a cardinality of (2..*). Hence, the constraint applies the following conditions:

- An object that plays role B must inherit from an object playing role A.
- There must be multiple objects (2..*) playing role B.

Furthermore, we annotate each role of the design pattern with its corresponding *feature role*. This means, each role played by a feature-oriented role is annotated with a role played by a feature. We describe the collaborations of these feature roles in a second role model. Here, we require constraints that express the necessary collaborations between features. We reuse the constraints *role prohibition and role equivalence* (cf. Section 2.3.1). In the context of features, *role prohibition* constrains a feature to play two roles in the same context and *role equivalence* expresses that these two feature roles have to be played by the same feature. To describe feature collaborations as generally as possible disregarding definite feature models, we also introduce the following constraint:

Requires (cf. Figure 3.2b)
To express dependencies between features (i.e., parent-child relations or cross-tree constraints), we introduce the *requires* relation between features. A feature playing feature role A requires a feature playing feature role B to be selected in the configuration.

In Figure 3.3a, we illustrate the collaborations of roles of the strategy pattern using a role model. This way, we can describe the design pattern in a general fashion disregarding a definite (object-oriented) design. We already described this role model in Section 2.3.2. In addition, we annotate each role with its corresponding feature role, whose collaborations we describe in Figure 3.3b. In particular, in this example, F_1 requires F_2 because there has to be a call from the *clientMethod* to the *strategyMethod*. F_1 and F_3 as well as F_2 and F_5 have to be played by the same feature, respectively, because the *clientMethod* and *strategyMethod* must be introduced by their corresponding feature-oriented roles. However, the two client roles *MethodClient* and *FieldClient*, introduced using features



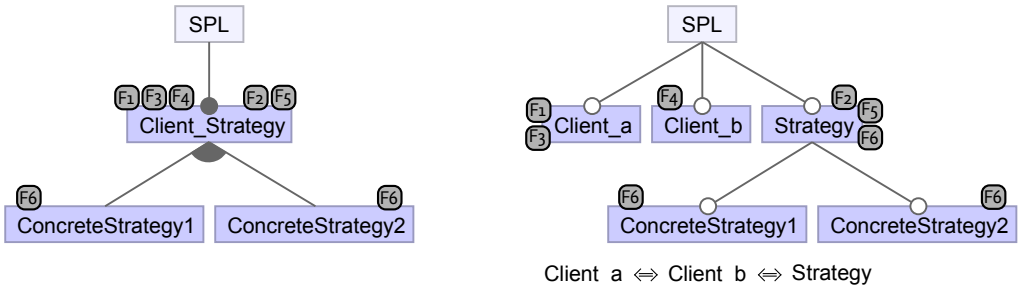
(a) Role model for the strategy pattern annotated with features

(b) Role model for participating features of the strategy pattern

Figure 3.3: Family role model for the strategy pattern

F3 and *F4* may not be introduced using the same feature, but always have to occur together in order for them to constitute the client. By holding a reference to the *Strategy*, the *FieldClient*, introduced in *F4* requires the *Strategy*'s feature *F5*. In order for the strategy pattern to hold, both *Strategy* and *Client* have to exist, thus *F4* and *F5* must be selected together. A *ConcreteStrategy* requires the *Strategy*, thus *F6* requires *F5*. For a more detailed description of this pattern, we refer to our results in Chapter 7.

Because this role model is a general description of feature collaborations, we can create different feature models that conform to this role model. We show two different feature models for this FRM in Figure 3.4. Note that we only illustrate extracts of feature models with feature relevant to the specific design pattern. In both feature models, we annotate each feature with the feature roles it is playing. For instance, in Figure 3.4a, the feature *Client_Strategy* plays all the feature roles *F1* to *F5*. Only the concrete strategies are introduced in their own features. In Figure 3.4b, on the other hand, there are two feature *Client_a* and *Client_b*, playing the different client roles *F1* and *F3* as well as *F4*, respectively. The *Strategy* feature plays the feature roles *F2*, *F5* and also *F6*, introducing a concrete strategy. More concrete strategies are added by its subfeatures. Many other feature models that conform to this FRM are possible, hence, with FRMs, we realize a general description of feature collaborations for a variability-aware design pattern.



(a) Feature model A

(b) Feature model B

Figure 3.4: Feature models extracts for variability-aware strategy pattern

3.4 Variability-Aware Design Pattern Catalog

By introducing design patterns for object-oriented software, Gamma et al. [16] changed the way we view software design. They documented these design patterns in a design pattern catalog using a consistent format for each design pattern (cf. Section 2.1). For descriptions of new design pat-

terns [8, 15, 42], this consistent format has often been adopted. Based on the catalog of Gamma et al. [16], we introduce a variability-aware design pattern catalog where we take feature semantics and the decomposition of patterns along feature roles into account. To this end, we use automated family-based design pattern detection to capture decomposed design patterns in existing feature-oriented product lines and analyze these patterns regarding their decomposition along features. From these results, we derive guidelines and application rules for variability-aware design patterns. In Figure 3.5, we exemplary depict the catalog page for the variability-aware application of the strategy pattern. For further information on this catalog page, we refer to our results in Chapter 7.

Based on the consistent format for design pattern description of Gamma et al. [16] and our notation for family role models (cf. Section 3.3), we describe design patterns regarding the following aspects:

Intent
: With the intent, we describe the application domain and motivation of applying this design pattern in the context of SPLs.

Solution
: The solution describes the guidelines and application rules, thus, the structure and interaction of feature-oriented roles and features using a FRM, providing a template solution to be adapt to specific application scenarios.

Consequences
: The consequences describe benefits and drawbacks of applying this pattern in the proposed fashion in the context of SPLs. This includes impact on the reuse, variability and modularity of the product line.

However, we do not describe general, object-oriented aspects of the solution and consequences as we focus on the variability-aware implementation of design patterns.

3.5 Summary

In this chapter, we introduced the idea of variability-aware design patterns as a concept to combine FOP with established object-oriented design concepts. We argue, that design patterns and FOP can coexist in symbiosis, affecting each other in a positive manner. To support this, we aim at revealing the implementation of design patterns in FOP case studies. Using a case study, we aim at detecting decomposed instances of design patterns to reveal their application in the context of SPLs.

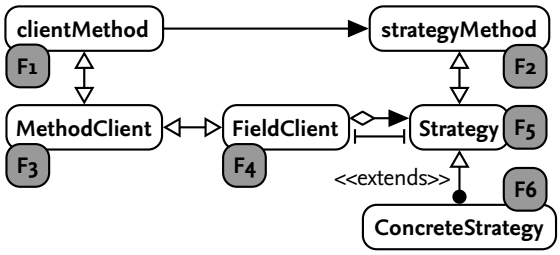
We apply role modeling in order to describe general collaborations of design patterns instead of a definite design. Using these role models, we are able to develop an automated, family-based design pattern detection technique for feature-oriented SPLs that we use to reveal the decomposed application of patterns. In order to develop this detection technique, we develop a system representation of feature-oriented Java code containing all necessary variability information. As the main contribution of this chapter, we introduced *family role models (FRM)* as a means to describe the decomposed application of design patterns. Based on these FRM, we create a variability-aware design pattern catalog containing guidelines and application rules of specific design patterns in the context of feature-oriented SPLs.

Strategy pattern

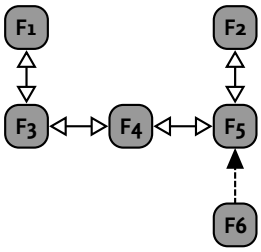
Intent

Encapsulate interchangeable algorithms/behavior while allowing compile-time selection.

Solution



(a) Role model for the strategy pattern annotated with features



(b) Role model for participating features of the strategy pattern

We depict the structure of the decomposed pattern above. In Figure 7.4a, we show the different roles with their introducing features, whereas in Figure 7.4b, we illustrate the relationships and dependencies of these features. We make the following suggestions:

- Introduce *Client* and *Strategy* of Figure 7.4a together in one feature. Hence, *F3* and *F4* as well as *F5* in Figure 7.4b are equivalent, which also implies the equivalence of *F1* and *F2*.
- Introduce *ConcreteStrategies* separately while requiring the *Strategy*.
- Make at least one *ConcreteStrategy* mandatory if *Client* and *Strategy* are introduced.
- Introduce *Client* and *Strategy* as well as *ConcreteStrategies* in the same feature group (i.e., common parent feature).

Consequences

Using this solution, the following consequences are implied:

+ Configure a *Client* at compile-time with varying behavior using one or more *ConcreteStrategies*.
+ Ease extensibility for new *ConcreteStrategies*.
– *Client* must be made aware of existing *ConcreteStrategies*. To this end, the feature template method (cf. Section 2.5) might be suitable, filling an operation to register strategies with each feature introducing a *ConcreteStrategy*.

Figure 3.5: Catalog page for the strategy pattern as an example for the variability-aware design pattern catalog

4 FOPJaMoPP – FOP goes EMF

In this chapter, we first motivate developing a variability-aware system representation and briefly explain our approach in Section 4.1, after which we introduce JaMoPP, the EMF-based *Java Model Parser and Printer* in Section 4.2. Next, we describe and illustrate the challenges of extending a Java parser for parsing FOP code correctly in Section 4.3. In Section 4.4, we give an overview on our implementation of FOPJaMoPP, our extension of JaMoPP capable of parsing feature-oriented Java code. In Section 4.5, we depict and explain the resulting, variability-aware system representation, before we finally give a brief summary on this chapter in Section 4.6.

4.1 Motivation

In order to run an automated family-based design pattern detection on FOP code, we need a representation of the code, which we can analyze. To this end, a structured data model is necessary, such as an *abstract syntax tree (AST)*. In order to perform a family-based analysis, we need the AST to also contain the variability information of the product line, meaning, which feature contributes to which class. We call this a *150% AST* in reference to the notion of a 150% model with annotative variability realization mechanisms. In our previous work [37], we used Fuji to create the AST which our design pattern detection was based on (cf. Section 2.5). However, as we explained in Section 3.2, the Fuji AST is impractical for family-based analyses.

Therefore, we decided to neglect Fuji for our purposes and developed our own parser for feature-oriented Java code based on the *Eclipse Modeling Framework (EMF)*, (cf. Section 2.4). Our decision was supported by the fact that with the *Java model parser and printer* (JaMoPP), a fully-fledged, EMF-based Java metamodel and parser already exists, which gives us the advantage of not having to model and parse complete Java code manually. However, simply using a Java parser to parse feature-oriented Java code is not sufficient as FOP allows us to refine classes. This means, a class can consist of several feature-oriented roles, which are parts of this class defined in various features. These feature-oriented roles can, for instance, reference elements, such as fields and methods, of other feature-oriented roles contributing to the same class. A regular Java parser cannot resolve such *inter-feature references* because, in its point of view, Java files containing inter-feature references are *illegal* Java files. In Section 4.3, we describe all challenges of parsing feature-oriented Java code using a regular Java parser, such as JaMoPP. Because of these, we extended JaMoPP to be able to handle feature-oriented Java code.

4.2 JaMoPP – Java Model Parser and Printer

Beyond other application scenarios, EMF is widely employed as the foundation for programming languages and domain-specific languages. Heidenreich et al. [18] developed an Ecore metamodel (cf. Section 2.4), an EMFText syntax and a static semantics analysis for the Java programming

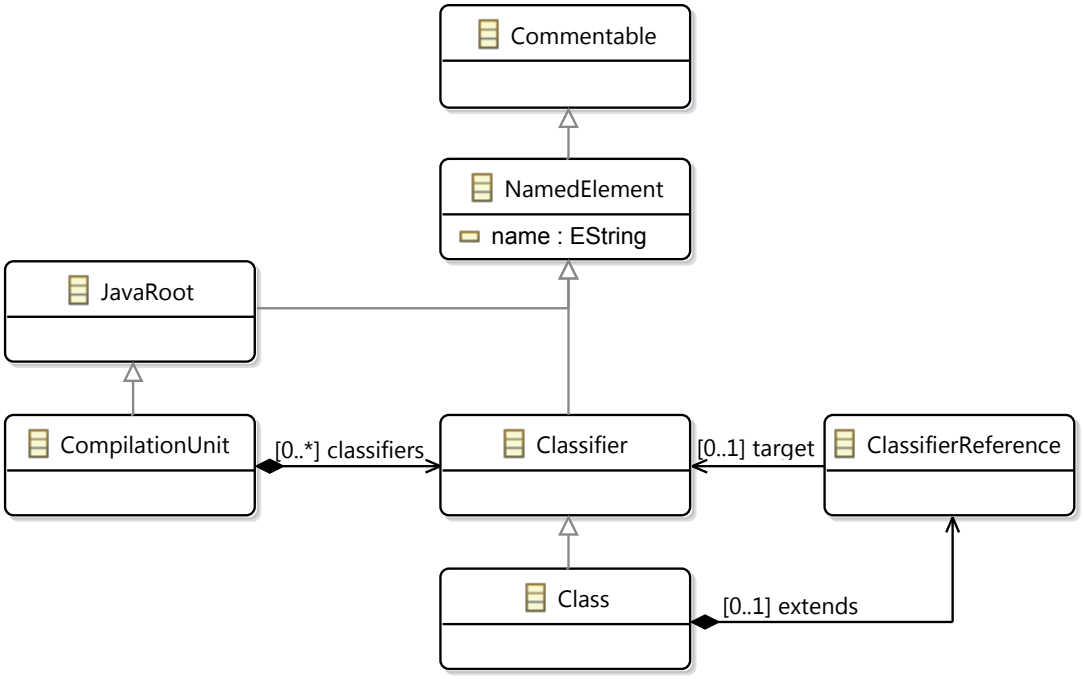


Figure 4.1: Simplified extract of JaMoPP metamodel

language, which resulted in the Java model parser and printer – or short: JaMoPP[1]. JaMoPP offers a fully-fledged metamodel and parser for Java 1.5.

In Figure 4.1, we depict an extract of the JaMoPP metamodel for the basic constructs in Java. We describe the elements of Ecore metamodels in Section 2.4. To demonstrate the concepts used in the JaMoPP metamodel, only for this description, we leave out unimportant parts and relationships in this extract. On the left hand side, we can see the `CompilationUnit` class, which is the root element of each Java file and extends the `JavaRoot` class. In Java, a compilation unit holds an arbitrary number of classifiers. This is modeled in Ecore using a containment reference between `CompilationUnit` and `Classifier` with cardinality $[0..*]$. Both, compilation units and classifiers (such as classes and interfaces) must have a name, hence, both extend the class `NamedElement`, holding an attribute `name` of type `EString`. Classes in Java can extend other classes using inheritance. Thus, classes hold a reference to the class they extend.

Especially relevant for this work is how references to elements, such as variables or members as well as classifiers are modeled in JaMoPP (cf. Section 4.3). Figure 4.1 includes the *extends* relation of classes in Java, which exemplary shows JaMoPP's way of modeling such references. Particularly, a `Class` object holds a unary containment reference to a `ClassifierReference` object. This way, classifier references are unique and bound to their parent class. This `ClassifierReference` object now holds a unary reference to its target. This is, of course, not a containment reference, because classes can exist without being referenced anywhere.

Every reference in JaMoPP is implemented this way. A `MethodCall` object is contained by its calling `Expression` and contains `ClassifierReferences` as arguments. The unary target reference to its declaring `Method` is non-containing. An `IdentifierReferences`, such as a reference to a

[1] Java model parser and printer (JaMoPP): http://www.jamopp.org/

Feature *Add*

```
1  import java.util.ArrayList;
2  class MyList extends ArrayList {
3    void myAdd(Object o) {
4      super.add(o);
5    }
6  }
```

Feature *Remove*

```
7
8  class MyList {
9    void myRemove(Object o) {
10     super.remove(o);
11   }
12 }
```

Feature *AddPrint*

```
13
14 class MyList {
15   void myAdd(Object o) {
16     original(o);
17     System.out.println("add: "+o);
18   }
19 }
20
```

Feature *AddAll*

```
21
22 class MyList {
23   void addAll(ArrayList l) {
24     for(Object o : l) {
25       myAdd(o);
26     }
27   }
28 }
```

Figure 4.2: Example of peculiarities in feature-oriented Java code

variable or field, is also contained by its calling `Expression` and holds a target reference to its declaring `Identifier`.

JaMoPP is also capable of resolving such references correctly in legal Java files using its semantic analysis. However, these reference resolvers can only deal with legal Java files. Therefore, in the next section, we explain the challenges of adapting such resolvers to feature-oriented Java code and describe how we solved these challenges.

4.3 Challenges of Parsing Feature-Oriented Java Code

As we explained in Section 2.2.2, feature-oriented Java code differs from common Java code in a few, yet crucial aspects. Since we employ JaMoPP, a parser that is capable of parsing and managing Java 1.5 syntax, we need to extend parts of JaMoPP in order to let it parse feature-oriented Java code correctly. In Figure 4.2, we illustrate an example code snippet of common feature-oriented Java code, exemplary showing the peculiarities of FOP. In the following, we use this example to express the challenges of parsing feature-oriented code.

Our goal is to be able to construct a 150% AST of feature-oriented Java code. This means, on the one hand, that we want to store information about which feature-oriented role (with its members) is introduced in which feature. To this end, we need to annotate the feature information to each parsed Java file. For instance, in Figure 4.2, when we parse the class `MyAdd` of feature *Add*, we want it to contain the name of the feature it is introduced in.

On the other hand, we have to take care of references to elements or classifiers. Because multiple feature-oriented roles can contribute to one class and refine elements defined in other feature-oriented roles, reference resolving for feature-oriented Java code is not as straightforward as for Java code because we are dealing with an enlarged scope for these references. When referencing elements, they can be declared in other features, thus making these references *inter-feature references*. An inter-feature reference is a reference to an element within one feature, whereas the declaration

of this element is performed in another feature. Moreover, these elements can be declared multiple times, for instance, if we refine a method in another feature, or if we have alternative features, both requiring a particular field, yet with different types. In order to perform a family-based analysis, and within that analysis, for instance, check whether a method calls a specific other method, we need to resolve element references to *all* possible targets of that reference. In particular, this means, that a reference to an element or a classifier should be resolved multiple times to *all* declarations of that element or classifier in *all* features. Resolving these inter-feature references is made even more complex by the fact that in FOP, common parts of a class only have to be specified in at least one feature-oriented role contributing to that class. In particular, this regards imports and supertypes. Moreover, we can use an `original` call in FOP during a refinement of a method to reference the method that we refine.

Of course, we want the Java reference resolver itself to do most of the work, so that we only have to concern ourself with references that the Java reference resolver cannot handle. To this end, we identified all peculiarities of feature-oriented Java code that a common Java reference resolver cannot take care of. In the following, we describe these peculiarities and our ways of solving these issues.

4.3.1 Imports across Feature-Oriented Roles

In Figure 4.2, we can identify the first peculiarity of feature-oriented Java code. Because different feature-oriented roles contribute to the same Java class, a necessary import only has to be provided in at least one of these feature-oriented roles, but still can be used in any of the feature-oriented roles contributing to the same class. Therefore, in the example, it is sufficient to import `java.util.ArrayList` only within one of the features (e.g., *Add*), while using it in other features such as *AddAll*. When the Java parser now tries to parse the feature-oriented role `MyList` of feature *AddAll*, it would fail to resolve the reference of the list `l`, because it is unable to locate the `ArrayList` class. However, we do not want to take care of resolving such references to imports manually, because a common Java reference resolver is capable of doing so if it has the necessary information of which imports exist. Thus, before resolving the reference, we need to capture the imports of other feature-oriented roles.

Our approach of solving this issue is quite straightforward, yet not universally applicable. In our approach, we simply collect all the imports of all feature-oriented roles contributing to the same class. Afterwards, we copy these imports into each of the feature-oriented roles. This way, we have all necessary imports of all feature-oriented roles within each Java file. Thus, the parser will always find an import for each classifier reference that needs importing.

This approach is not universally applicable, and in Figure 4.3, we illustrate the reason. The problem lies within the way we can alternatively use different imports in different features for a similar classifier. In the example, we have two alternative features *PC* and *Mobile*. Both of these features have to handle objects of type `Image`. However, they use different `Image` implementations depending on the platform and, consequently, import different `Image` types. With our approach, both feature-oriented roles of `ImageUser` in both features, *PC* and *Mobile*, would get a copy of the other feature-oriented role's imports. Hence, both feature-oriented roles would have two imports of type `Image`, which is not valid in Java. While we can ignore the invalidity of our files because JaMoPP does not check whether the static semantics of the parsed Java files are valid, we face a problem

Feature *PC*

```
1  import java.awt.image;
2  class ImageUser {
3   void draw() {
4     Image i = new Image();
5     /* ... */
6     i.getWidth();
7     /* ... */
8   }
9  }
```

Feature *Mobile*

```
10 import javax.microedition.lcdiu.Image;
11 class ImageUser {
12  void draw() {
13    Image i = new Image();
14    /* ... */
15    i.getHeight();
16    /* ... */
17  }
18 }
```

Figure 4.3: Parsing FOP – Failing when collecting imports

here. When we try to resolve the reference to `Image`, we would have to guess, which of both `Image` types is employed. In our approach, we simply take the first import that fits our referenced type by name, thus, in this case, we would target both references to `Image` to `java.util.Image`. However, `java.util.Image` does not contain a method `getHeight`, which means we would fail to resolve the reference of the `getHeight` call in Line 15.

While this leads to an approach that is not generally applicable, we argue that there is no universal answer. If we take the `getWidth` method in Line 6, for example, both types of `Image` offer a method fitting to the required signature. Thus, if we were to resolve this reference, we can only guess, which of the `Image` types is referenced here. The reason for this problem lies within the fact that we are conducting a family-based analysis. If we were to conduct a product-based analysis, such cases could not occur. Moreover, when considering the feature model and the feature order, one might be able to work around this issue in a family-based approach. Still, there might be cases when even that would not be sufficient. For instance, considering the example in Figure 4.3, if we have a feature that is based on the fact that both `Image` types offer a method with the same signature, such as `getWidth`, and this feature does not depend on other features, we would still have to guess the correct type. Nevertheless, this straightforward approach satisfies our requirements since none of our considered feature-oriented product lines exploits the ability of alternatively importing different types with the same name in different features. Thus, we are able to resolve every reference to imported types in all relevant cases.

4.3.2 Supertypes across Feature-Oriented Roles

Similar to imports, supertypes of classes in FOP only have to be provided within at least one feature-oriented role. We can see this in Figure 4.2, where we define the *extends* relation in Line 2 in feature *Add* and reference the super type in Line 10 in feature *Remove*. A Java parser cannot resolve this *super* reference.

In order to let a Java reference resolver handle references to the supertypes, we need to perform a similar approach to supertypes as to imports. We iterate through all the feature-oriented roles contributing to the same class and store the *extends* relation specified by at least one of these feature-oriented roles. We then copy it to all feature-oriented roles. Moreover, for implemented interfaces, we collect all *implements* references from all feature-oriented roles contributing to the same class and copy this set of *implements* references to all considered feature-oriented roles.

Feature *Array*

```
1  import java.util.ArrayList;
2  class MyList extends ArrayList {
3    void myAdd(Object o) {
4      super.add(o);
5    }
6  }
```

Feature *Linked*

```
7  import java.util.LinkedList;
8  class MyList extends LinkedList {
9    void myRemove(Object o) {
10     super.remove(o);
11   }
12 }
```

Figure 4.4: Parsing FOP – Failing when collecting supertypes

However, simply storing the extends type and collecting the implements types can lead to similar problems as we face with the imports, thus, making it a non-universal approach. In Figure 4.4, we depict an example where we fail to correctly resolve references to the supertype. This happens, for example, if we specify different supertypes in different features for the same class. In Line 2 in feature *Array*, for instance, we introduce a class with the supertype `ArrayList`, whereas, in Line 8 of feature *Linked*, we change this supertype to be `LinkedList`. If we now simply *take* the supertype of each feature-oriented role and afterwards copy it to all feature-oriented roles, we would have to decide, which one to take. In our case, we simply keep the last *extends* reference and copy that to all feature-oriented roles. However, this way, we resolve the super reference in feature *Array* to `LinkedList`.

Similar to imports, there is no way of telling which reference is correct without concerning the feature model and the feature order. Still, for our purposes, the pragmatical approach of simply collecting appears to be sufficient for all considered cases.

4.3.3 Resolve Inter-Feature Element References

Now that we have the necessary information about imports and supertypes in each feature-oriented role, all the references to classifiers can be resolved by a Java reference resolver. However, when we resolve a reference to a class that is declared with multiple feature-oriented roles, the reference resolver can only target one of these feature-oriented roles, which is picked arbitrarily. We have to consider this fact in the following.

In contrast to classifier references, we are not yet able to resolve all the references to elements. Element references are references to methods as well as identifiers (i.e., fields and variables). In order to reference an element in Java, we can define the location of this element using the Java dot notation by means of the dot operator (`'.'`). This way, we can chain element references, and, using the dot, gain access to elements within the type of the latter element reference. We consider the following example:

```
foo.bar().baz
```

We access the locally declared element `foo` directly, and, using the dot operator, we can access the elements declared by the type of `foo`, for instance, in this case, method `bar()`. Chaining dot operators, we can now access elements of the type of method `bar()`, for example, the field `baz`. In the following, when considering element references, we call the element on which they are called the *previous* reference (i.e., in this case, `foo` is the *previous* reference of `bar()`). Moreover, we call

element references called on another element reference the *next* reference of that reference (i.e., in this case, `bar()` is the *next* reference of `foo`).

We see that we can either reference a local element directly, or we can access an element defined by the type of another element. Therefore, scoping such elements in Java is quite straightforward.

However, in FOP, due to multiple feature-oriented roles contributing to the same class, a type is no longer unique. All of these feature-oriented roles are class declarations sharing the same qualified name of the class they contribute to, thus, all of these feature-oriented roles can be considered as the referenced type. This means, an element can be declared in various feature-oriented roles of the same class. To this end, for each element reference, we have to consider all the different feature-oriented roles of the type the reference is called on. We can see such an inter-feature reference in Figure 4.2. In this example, the method `myAdd` is declared multiple times in features *Add* and *AddPrint*. However, we call `myAdd` in Line 25, in feature *AddAll*. To resolve this method call, we now have to consider all other feature-oriented roles of class `MyList`.

In Java, the element cannot only be declared directly within the type it is called on, but also within one of its supertypes, or, if the type is an inner class, within the outer class. Moreover, we have to manually scope references to the *length* field of array, since arrays are not types of their own and cannot provide fields and methods.

In order to resolve such element references in FOP, we have to always consider the fact that classes are split into multiple feature-oriented roles. Therefore, we have to perform the following steps:

1. Figure out, on which type the element reference is called.
   - If a previous reference exists, use the type of that previous reference.
   - If no previous reference exists, use the type containing the reference.
2. Lookup the element in every feature-oriented role of that type.
3. Lookup the element in every feature-oriented role of every supertype of that type.
4. If the reference is called within an inner class, lookup the element in all roles of the outer role.
5. If the reference targets a *length* field of an array, manually scope to such a field.
6. Store all detected elements as possible targets for that reference.

Following these steps, we are able to resolve all existing element references across feature borders correctly. Moreover, we are able to gain access to *all* element declarations of an element by storing every detected element that fits to a particular reference. We decided to go with the approach of resolving references to multiple targets because it facilitates family-based code analysis on a structural level. The alternative would be to only store one element declaration belonging to a reference, for instance, depending on the feature order, storing only the first or the most recently declared one. However, then we would always have to manually detect every other existing element declaration, besides the one that is targeted by the reference, each time we come across an element reference. The latter approach would be much more time-consuming during analysis. Since we already have all the necessary information within the reference resolving, we can just as well store it.

4.3.4 Resolve `original`-call

In order to refine methods and, thus, be able to extend classes in a fine-grained manner, FeatureHouse offers the keyword `original` that is used within a refining method to call the refined method. We see the application of the `original` call in Line 16 in Figure 4.2. In this case, we use the `original`-call in feature *AddAll* to call the originally implemented method `myAdd` in feature *Add*.

The `original`-call is applied as a method call, having the signature of the refined method. It is therefore necessary to pass arguments of the respective types to the original method. Because of this, `original` is parsed by a Java parser as a method call, thus, an element reference to a method. Different solutions exist to resolve the `original`-call, for instance, based on the feature order we could figure out which method declaration is the most recent, and target that one. However, in our approach of resolving method calls, we chose to target all existing method declarations for that call, resolving the reference multiple times.

Because we are aiming at a family-based analysis of feature-oriented code, resolving to *each* method declaration, the one introducing the method and all the refining ones, would be most suitable. This way, we store all the available information, which is the most reusable solution. It can also be applied for a product-based analysis, taking the feature order and the feature model into account, filtering the necessary method declarations.

However, we face one major drawback using this approach. Because the `original`-call is parsed as an ordinary method call and we resolve it to all method declarations of the refined method, in the 150% AST, we cannot differentiate between an `original`-call and a recursive call of the method. Both cases would look exactly the same, a method call targeting all method calls of the containing method. Nevertheless, there would be a solution for this issue that could possibly be implemented without much effort. We could introduce a new class in the metamodel `OriginalCall` and adapt the parser rules such that the `OriginalCall` is an alternative to a `MethodCall`. Moreover, we adapt the parser rules by first checking whether the identifier of a `MethodCall` is "original". If this is the case, we can parse it as an `OriginalCall`. This way, we would be able do distinguish between `original`-calls and recursive method calls.

We did not implement this extension to the parser because, we argue, while this is a major drawback for general code analyses, it does not hinder the structural analysis that we conduct. Hence, we argue, that for our purposes we can ignore this issue. However, for other analyses, such as control flow analyses or explicit analyses of refinements, are planned, this issue would have to be tackled.

4.4 Implementation

In the last section, we described the requirements of parsing feature-oriented Java code. With our tool FOPJaMoPP, which is an extension to JaMoPP (cf. Section 4.2), we solved these aforementioned challenges of creating a 150% AST. To this end, we developed tooling regarding imports and supertypes of feature-oriented roles (cf. Section 4.3.1 and Section 4.3.2) as well as a feature-oriented reference resolver, taking feature semantics into account (cf. Section 4.3.3 and Section 4.3.4). To realize multi-target references, as we need them to store the targets of element references, we extended the metamodel of JaMoPP by a few, yet necessary classes.

In the following, we explain our implementation of FOPJaMoPP by first describing the development environment. Next, we depict the rough architecture. After that, we introduce the FOP-

JaMoPP metamodel, which is the required extension of the JaMoPP metamodel necessary to represent multi-target references of feature-oriented code. Then, we focus on the different steps of the workflow, where we briefly describe how exactly we collect and distribute imports and supertypes and the technical peculiarities of developing the extended reference resolver.

4.4.1 Environment

As JaMoPP, we implement FOPJaMoPP as a plugin for Eclipse[2]. For the transformation to our extended metamodel, we use Xtend[3], a Java dialect offering powerful macros that ease model traversal. We use the following versions:

- Java Runtime Environment (JRE) 7
- Eclipse Modeling Tools – Kepler SR2 (4.3.2)
- JaMoPP 1.4.0
- Xtend 2.6.0

4.4.2 Architecture

FOPJaMoPP is an add-on to JaMoPP, consisting of only two plugins that provide the basic functionality as well as an extension of the reference resolvers of JaMoPP, which had to be placed within the particular JaMoPP plugin:

`org.emftext.language.java.resource.java`
: Within this JaMoPP project, we extend the JaMoPP `ElementReferenceTargetReferenceResolver` with the functionality to resolve inter-feature element references. To this end, we introduced the `FopElementReferenceTargetReferenceResolver` that is used in addition to the regular element reference resolver to resolve inter-feature element references. Moreover, we introduce the helper class `FopElementReferenceResolverUtil` that is used to manage and store the different feature-oriented roles and the multi-target references as well as collecting the supertypes across feature-oriented roles contributing to one class (cf. Section 4.3.2).

`de.tu-bs.cs.isf.fopjamopp`
: Within this plugin project, we provide the extended FOPJaMoPP metamodel and the corresponding, generated model code, as well as the basic functionality of FOPJaMoPP consisting of different parts divided into the following packages:

  importcollector
  : The `FopJavaImportCollector` realizes the import collection as we described in Section 4.3.1.

  importer
  : The `importer` package consists of two parts. First, the `FopJavaImporter` triggers the process of FOPJaMoPP by loading the feature-oriented Java files as resources and afterwards starting the workflow as we describe in Section 4.4.4, finishing using the below

[2] Eclipse IDE: http://www.eclipse.org
[3] Xtend programming language: http://www.eclipse.org/xtend/

described `XmiExporter` and serializing the parsed and resolved feature-oriented Java files. Secondly, to start an analysis on the feature-oriented code, that is now serialized as XMI files, we also offer the `XmiImporter`.

exporter
In the `exporter` package, we provide the `XmiExporter` that is used to serialize loaded and resolved, feature-oriented Java files, thus, the 150% AST, as XMI files.

converter
As described above, we need to extend the JaMoPP metamodel with a FOPJaMoPP metamodel to store the multi-target references within the AST. The `converter` package holds the `JamoppToFopjamoppConverter` that is used in the last step of FOPJaMoPP, converting the AST to a 150% AST as we need it. Moreover, we provide the `FeatureNameAnnotator` that annotates each compilation unit of each feature-oriented Java file with its corresponding feature name.

`de.tu-bs.cs.isf.fopjamopp.ui`
Within this plugin project, we extend the Eclipse user interface with menu items for starting FOPJaMoPP on an existing project containing FeatureHouse code.

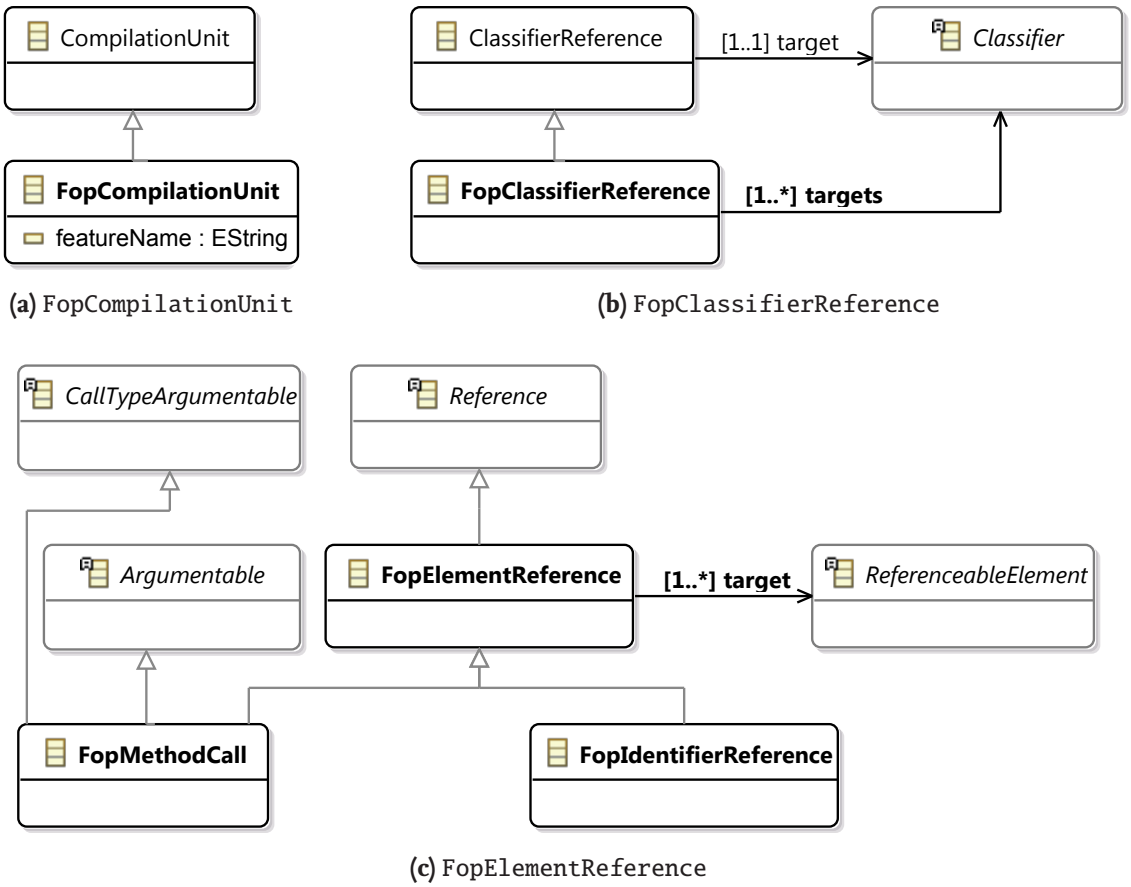
4.4.3 FOPJaMoPP Metamodel

In order to store the feature for each Java file and to store multi-target references in our 150% AST, we extend the JaMoPP metamodel in a few, yet crucial aspects. In Figure 4.5, we illustrate the extensions of the JaMoPP metamodel resulting in the FOPJaMoPP metamodel. Each newly introduced class starts with *Fop*, all the other ones are classes introduced by JaMoPP.

In Figure 4.5a, we show the necessary extension to store the feature name for each parsed feature-oriented Java file. The root object of each file, `CompilationUnit`, is extended by the `FopCompilationUnit`, adding the attribute `featureName` of type `EString`. By inheriting from `CompilationUnit`, we can simply exchange every parsed `CompilationUnit` object with `FopCompilationUnits`, adding the feature attribute to each file.

In Figure 4.5b, we depict the necessary extension to store multiple targets for a `ClassifierReference`. We extend the `ClassifierReference` class with `FopClassifierReference`, adding a new multi-reference with name `targets` to `Classifier`. This way, we can exchange all objects of type `ClassifierReference` with a `FopClassifierReference`, and add all feature-oriented roles of the targeted `Classifier` object. We needed `FopClassifierReference` to be of type `ClassifierReference` because there are references from other classes, for instance, `NamespaceClassifierReference` targeting `ClassifierReference`. One drawback of this approach is, however, that we now have objects of type `FopClassifierReference` with an empty attribute `target`.

The last extension, regarding element references, that we illustrate in Figure 4.5c, is a bit more complex. In JaMoPP, element references such as `MethodCall` and `IdentifierReference`, share the supertype `ElementReference`, which is a subtype of `Reference` and adds the `target` reference to a `ReferenceableElement`. To extend this `ElementReference` with a multi-target reference, we construct a parallel structure, subtyping `Reference` with `FopElementReference`, subtyped by `FopMethodCall` and `FopIdentifierReference`. We do not need our `FopElementReference` to be of type `ElementReference` since there are no direct references to `ElementReference`, but rather to



(a) FopCompilationUnit (b) FopClassifierReference

(c) FopElementReference

Figure 4.5: FOPJaMoPP – Extensions to JaMoPP metamodel

Reference. To exactly resemble the MethodCall, we must not forget to make it Argumentable and CallTypeArgumentable, which adds the possibility of adding arguments and generics, respectively. Using this approach, we can exchange all MethodCall and IdentifierReference objects with FopMethodCall and FopIdentifierReference, respectively, storing the various targets of each reference.

4.4.4 Workflow of FOPJaMoPP

FOPJaMoPP consists of several steps that are necessary to parse feature-oriented Java code. Since FOPJaMoPP is designed as an add-on to JaMoPP, we did not change JaMoPP besides plugging in our own reference resolvers as we described in Section 4.4.2. In Figure 4.6, we depict the workflow that illustrates how FOPJaMoPP works, containing the necessary steps to gain a model representation of feature-oriented Java code.

Running JaMoPP on Feature-Oriented Java Code

We start with running JaMoPP in order to load and parse all feature-oriented Java files. JaMoPP simply parses each Java file to its AST without checking the uniqueness of the qualified name or whether the file is legal. Moreover, JaMoPP does not resolve references, but rather stores proxies containing information about the particular reference. Already in this step, we annotate all Java files with their corresponding feature name, by exchanging each object of type CompilationUnit

Figure 4.6: FOPJaMoPP – Workflow

with an object of type `FopCompilationUnit`, that we introduced in our FOPJaMoPP metamodel in Section 4.4.3. We can compute the feature name based on the path of each file. Using this AST, we conduct the necessary steps in order to gain a 150% AST of feature-oriented code.

Collect Imports across Feature-Oriented Roles

We developed the `FopJavaImportCollector` that we mentioned in Section 4.4.2 based on our approach on collecting imports across feature-oriented roles explained in Section 4.3.1. To this end, we first collect all objects of type `CompilationUnit`, which are all existing Java files in the project. We map each compilation unit to its qualified name, thus, gaining a map of qualified names and the corresponding feature-oriented roles contributing to the same class. For each qualified name, we now collect all existing imports while removing duplicates and afterwards copy these imports to all corresponding feature-oriented roles.

Having all necessary imports within each feature-oriented role, lets us reuse the complete classifier reference resolver of JaMoPP, which is able to resolve classifiers based on the existing imports. Without this step, we would have to resolve not imported classifiers ourself.

Collect Superclasses and Interfaces across Feature-Oriented Roles

Similar to collecting the imports, before resolving the references in the next step, we take care of the supertypes of different feature-oriented roles contributing to the same class. As we described in Section 4.3, in FOP, supertypes of a class only have to be defined in at least one feature-oriented role contributing to that class. To reuse most of the Java element reference resolver, we aim at having all possible supertypes for each class within each feature-oriented role contributing to that class. Since supertypes are not bound to compilation units, but to classifiers, and each compilation unit can contain several classifiers, we collect all feature-oriented roles of all classifiers of our product line and map them to their qualified name. We then collect the supertypes and copy the collection back to each feature-oriented role, as we described in Section 4.3.2.

This step is triggered by the extended element reference resolver right before trying to resolve a reference and is located within the `FopElementReferenceResolverUtil` in the JaMoPP plugin `org.emftext.language.java.resource.java` (cf. Section 4.4.2).

Resolve Inter-Feature Element References and the `original`-call

The next step is to actually resolve all element references. To this end, our `FopElementReferenceTargetReferenceResolver`, that we plugged into JaMoPP as the default element reference resolver, is triggered. However, the first step of resolving references is to call the original Java reference resolver of JaMoPP because it can handle most references. Afterwards, we run our extended reference resolver to also capture inter-feature references. For the basic concept of how our reference resolver works, we refer to our earlier description of the approach in Section 4.3.3. However, implementing the element reference resolver, we had to solve several technical challenges that we explain in the following.

Previous Reference

First, we needed to figure out, on which type the reference is called. While this sounds straightforward (i.e., simply compute the type of the *previous* reference or, if not existing, take the containing type), there are various difficulties. The previous reference can be of various types, for instance, an identifier or a method call, but also a classifier in case of static calls, as well as a constructor call, etc. Moreover, in rare cases when the previous reference is an identifier reference that points to a non-existing identifier, the JaMoPP parser is not able to identify it as an identifier reference. Thus, it is parsed as a `PackageReference`. We need to identify such cases and exchange that `PackageReference` with an `IdentifierReference`, which must be resolved before resolving the actual considered reference.

Method Calls

When resolving method calls, we need to check whether the arguments match the parameters of a specific method. To this end, we need to compare the type of an argument with the type of a parameter. However, this gets complicated when considering different feature-oriented roles of classes, leading to non-unique types. JaMoPP is not able to consider different feature-oriented roles of one class as the same type. Depending on the order in which the feature-oriented roles are parsed and the locations of the method call and the method declaration, different feature-oriented roles might be referenced as argument and parameter types. To check whether these types are equal (or the parameter is a supertype of the argument), we need to extend the mechanism of JaMoPP that makes these type checks. To this end, when comparing the argument types of a method call to the parameter types of a particular method declaration, we iterate through each pair of arguments and parameters and compare any of the feature-oriented roles of the argument type to the parsed parameter type. If we can identify one matching type, we go on to the next argument – parameter pair. If we can find one matching feature-oriented role for the argument type for each parameter, we target that method declaration in that method call.

`Array.length`

Because arrays are not specific types in Java, JaMoPP does not handle them as types. An array in JaMoPP is stored with its datatype and an additional attribute regarding the array's dimension. Thus, if the `length` attribute of an array is referenced, it cannot be resolved because that identifier does not exist within any datatype. The trick here is to identify the cases, when the `length` attribute is called on an array and to resolve it to a newly created field `length` that is added to that particular type.

`original-call`

Resolving the `original`-call is straightforward. If we try to resolve a method call with the identifier "original", we assume it is an `original`-call. We collect all method declarations of the method, which the `original`-call is contained by.

Since we have not yet transformed our model to the FOPJaMoPP metamodel because we still run the original JaMoPP reference resolver. As running that on our adapted model would lead to difficulties, we cannot yet store multi-target references within the respective reference objects. Our solution is to store the first detected declaration for each reference as the target of that reference

and to store all other targets within `FopElementReferenceResolverUtil` (cf. Section 4.4.2). This way, we can check whether we can resolve every reference (i.e., no more proxies), but still have all the multi-target references stored for the next step.

Transform JaMoPP metamodel to FOPJaMoPP metamodel

As described before, we have now resolved all references to one particular target and stored all other targets in `FopElementReferenceResolverUtil`. Now it is time to transform the model to our own FOPJaMoPP metamodel, supporting multi-target references. This step is quite straightforward, but was still tedious to implement. In order to exchange all element and classifier references with our feature-oriented pendants, we need to place our new reference objects in the exact location where the original reference was located. However, in Java, element reference can be contained within any kind of expression, for instance, a nested or a conditional expression, all of them having different characteristics. Thus, in order to exchange every element reference with its feature-oriented representations, we needed to consider every single kind of expression and, for every expression type, change the respective containment reference to the new, feature-oriented element reference. With 34 different expression types, one could imagine the work going into this step. Classifier references only have to be exchanged in eight different locations, such as cast expression or variable declarations as well as extends or implements references.

4.5 150% AST

After the execution of FOPJaMoPP, we gain a 150% AST of the feature-oriented product line that has been parsed, containing all variability information:

- Compilation units attributed with feature names
- Multi-target references for elements and classifiers
- Resolved `original`-call to multiple targets

In Figure 4.7, we illustrate an extract of such a 150% AST on the example of the feature-oriented role `GPL.Vertex` that is located in feature *BFS* of the *graph product line (GPL)*. The changes to the regular JaMoPP AST are highlighted. First, the `FopCompilationUnit` that replaced JaMoPP's `CompilationUnit` is attributed by the name of the containing feature, in this case *BFS*. Moreover, references to classifiers and elements are now multi-target references, such that we can target every declaration of every classifier or element. We can see a `FopClassifierReference` at location (2), which describes the implements relation of the feature-oriented role. Here, all feature-oriented roles of the implemented interfaces are referenced, such that the `EdgeIfc` is listed four times, for each feature it occurs in. The same kind of multi-target reference is featured for elements as well. In location (3), we see a `FopIdentifierReference` that targets multiple declarations of the field `visited` in features *BFS* and *DFS*. In location (4) we see the targets of an `original`-call. As we described, an `original`-call is parsed as a `MethodCall` and then replaced by a `FopMethodCall` targeting all the declarations of the containing method. In this case, the `FopMethodCall` targets all declarations of the method `display` in the feature-oriented roles contributing to the class `GPL.Vertex`, because the `original`-call is contained in one of these declarations of the `display` operations.

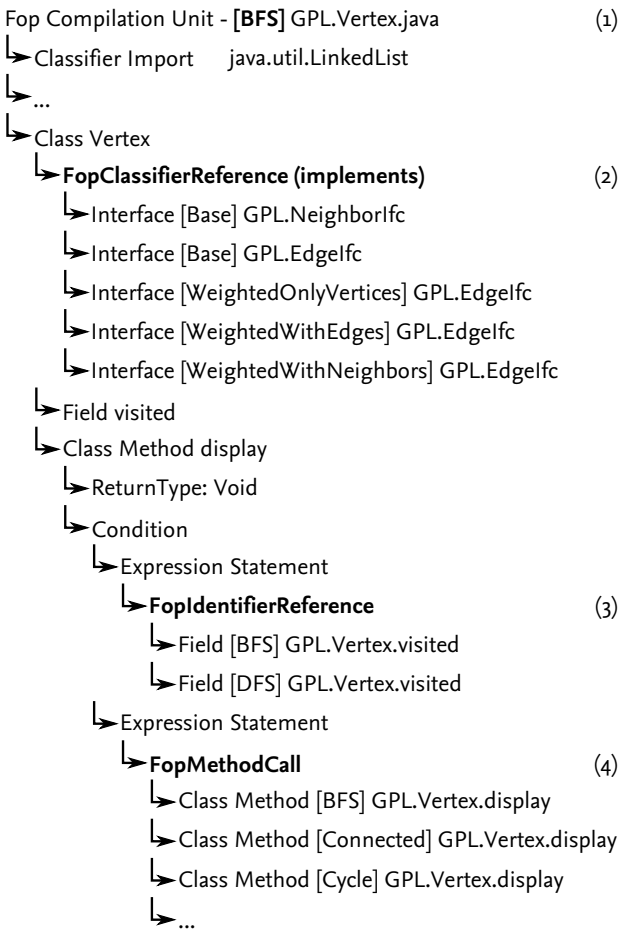


Figure 4.7: Extract of the 150% AST of the feature-oriented role `GPL.Vertex` of feature *BFS* in the GPL

4.6 Summary

In this chapter, we introduced FOPJaMoPP, a parser for feature-oriented Java code based on JaMoPP, the Java Model Parser and Printer, a reliable EMF-based parser for Java 1.5 code. FOPJaMoPP features a 150% AST, which is suitable for family-based analyses because it contains all necessary variability information (cf. Figure 4.7). Using the feature attribute of a `FopCompilationUnit`, which is the root element of each feature-oriented role, we can identify the enclosing feature for each element. Moreover, inter-feature references are resolved to all existing declarations of the referenced element across feature-oriented roles (i.e., multi-target references), such as class declarations (or refinements), method declarations or field declarations. The FeatureHouse specific `original`-call, which is used to call the original code of the refined method, is resolved to all method declarations of the refined method. Employing this 150% AST, we can now start all kinds of family-based analyses on feature-oriented Java code.

5 Family-Based Design Pattern Detection with Pattern Demon

Now that we constructed the 150% AST containing all variability information of the feature-oriented SPL, we can employ it as the system representation for the automated design pattern detection. Design pattern detection is an extensive research topic itself with Dong et al. [13] and Rasool et al. [29] both publishing survey papers on various approaches. Our goal is to develop a family-based approach that is capable of detecting design patterns across feature-oriented product lines.

In the following, we first motivate the development of a new family-based approach for automated design pattern detection in Section 5.1, followed by a brief review on existing approaches for object-oriented software in Section 5.2. In Section 5.3, we explain our approach in detail, after which we describe Pattern Demon, the implementation of our approach, in Section 5.4. Next, in Section 5.5, we illustrate the detection process for the four design patterns we want to detect in this work (cf. Section 3.2). Finally, we give a brief summary on this chapter in Section 5.6.

5.1 Motivation

In our previous work concerning design patterns in FOP [37], we already developed a structural detection approach based on the static analysis of Heuzeroth et al. [20]. In that work, we used manual AST traversal to identify certain class and object structures resembling design patterns. We based our analysis on the Fuji AST, which, in retrospect, is not suitable for family-based analysis (cf. Section 3.2).

Besides our previous work on this topic, no research on pattern detection has been done in the context of SPLs. Thus, we aim at extending our former approach to make it more efficient. We already achieved the first step of using EMF to represent our system by developing FOPJaMoPP. With EMF, we gain a system representation that provides us with automatically managed containment hierarchies, reference resolving and plenty of tool support, easing the implementation of a pattern detection approach. However, we also want to extend our detection approach, reusing existing approaches that were developed for standard stand-alone software. To this end, we reviewed different approaches of design pattern detection that have been published over the years. Because we develop a family-based approach, we are limited to static analyses and can, thus, only take existing static analyses into account for our detection approach.

In the following, we first describe existing design pattern detection techniques, after which we present our family-based detection approach that we developed based on our 150% AST created using FOPJaMoPP. Moreover, we explain the limitations of our detection technique. Furthermore, we illustrate the implementation of our approach, after which we describe how exactly we detect various design patterns, such as the template method pattern or the observer pattern (cf. Section 2.1.3).

5.2 Automated Design Pattern Detection

There are plenty of existing design pattern detection techniques as Dong et al. [13] and Rasool et al. [29] present in their surveys, using structural, data flow or control flow analyses or behavioral analyses by applying various concepts of different fields, for instance, graph matching, machine learning or data mining techniques. However, different design pattern detection techniques focus on different aspects of design patterns. Some are more suitable for detecting structural design patterns, others focus more on detecting interactions of objects at runtime and are better to detect behavioral design patterns (cf. Section 2.1.1). Hence, there is no ultimate technique for design pattern detection, but rather a set of techniques, whose suitability depends on the respective context [13]. Because we are limited to static analyses, we focus on describing existing static approaches.

Most static detection techniques for design patterns have in common that they describe structural aspects of patterns by defining inter-class collaborations, such as inheritance relations or method calls, that have to be met in order for a given structure to resemble a design pattern [13]. In addition, they combine this structural analysis with other, more sophisticated techniques, for instance, data and control flow analyses, scoring algorithms as well as dynamic or semantic analyses, to identify false positives (i.e., matches that are not design patterns) or to also identify variations of patterns instead of solely focusing on exact matches [13]. This way, they narrow down the set of matches in order to be as accurate as possible. Furthermore, most approaches differentiate in the system representation on which their approach works, such as the AST, an *abstract semantic graph (ASG)* or matrix and vector representations of the class and object relations [13].

Heuzeroth et al. [20], for instance, capture the specific characteristics of design patterns that describe the minimal structural requirements, which have to be fulfilled for a specific class and object structure to resemble the pattern. They developed algorithms for five design patterns, in particular, the *observer*, *composite*, *mediator*, *visitor* and *chain of responsibility* patterns, traversing the AST of a software system and incrementally checking the different structural requirements on each combination of classes and objects. In addition, Heuzeroth et al. [20] apply lightweight data and control flow analyses in order to be more precise in their static analysis. For example, they check whether a method calls another specific method within its body, or whether a method potentially stores its parameters. According to Heuzeroth et al. [20], they were faced with a number of false positives during their static analysis, which they decreased by subsequent behavioral analysis by checking constraints on object relations at runtime. Combining both structural and behavioral analysis they achieved to develop a reliable technique for design design pattern detection that is also able to capture behavioral patterns.

A piece of software can be expressed using a graph structure, which means, we have classes being nodes in the graph, and class inheritance as well as object composition resembling the edges. Tsantalis et al. [40] use graph pattern matching combined with a similarity scoring algorithm to identify graph structures that are exact representations or similar structures of a specific design pattern. By this means, the authors achieve a more general detection approach, also detecting pattern instances that do not resemble the exact structure exemplified by Gamma et al. in [16]. Especially concerning transitive relationships between classes, for instance, inheritance chains, is quite important in order to not only focus on one exact structure of the pattern, but rather focus on the required collaborations of objects and classes.

Other static design pattern detection approaches have similar structural detection techniques, using matrices or graphs and checking defined conditions on inter-class relationships. However, there are many ways to extend this structural analysis, as we already explained for Heuzeroth et al. [20] and Tsantalis et al. [40]. Shi et al. [38], for instance, define *control flow graphs (CFG)* for each method that is required for a design pattern, such as the `getFlyweight` operation of the *flyweight* pattern [16, p. 195 ff.] by defining which conditions and statements are necessary to fulfill the requirements for that particular method. For example, they specify that particular objects have to be defined or that a particular object has to be returned. This way, they can also detect behavioral aspects while using a static approach.

Hence, there are plenty of advanced and feasible techniques for automated design pattern detection, however, focusing on different aspects of design patterns. All of these approaches share similar techniques for structural detection combined with more advanced techniques to identify false positive matches. However, static approaches still suffer from inaccuracy leading to a number of false positives, which is why several techniques are combined with behavioral or semantic analyses in order to increase feasibility [29].

5.3 Approach

Because we concentrate on SPLs and on design patterns relevant for modular product line development, which are patterns that encourage modularization, encapsulation and reuse (cf. Section 3.2), we aim at developing an approach that is most suitable for these patterns, while disregarding other design patterns for now. Moreover, we reuse the general concept of detecting structural matches at first and afterwards eliminating false positives using advanced analyses.

A challenge of family-based design pattern detection is that a design pattern instance can be decomposed across several features, which means that parts of the pattern can be scattered across several feature-oriented roles. Thus, if the exemplary structure of a design pattern includes one class introducing multiple fields or operations, we have to consider that each of these members could be introduced within another feature. Moreover, refinements of such members is possible. Therefore, we have to consider the collaborations of feature-oriented roles instead of mere class collaborations. Role modeling (cf. Section 2.3) is the perfect tool to describe collaborations of objects of any kind and it has successfully been applied to design patterns (cf. Section 2.3.2). Thus, we apply role modeling in order to describe design patterns in detail and collaboration-based. This way, we can try to detect the different roles of a design pattern while matching these roles to feature-oriented roles and methods of the product line.

Moreover, we do not have an executable software available, but rather the 150%-model of our software product line. Therefore, we are limited to static analyses, such as structural, data flow or control flow analysis, since dynamic analyses would require a running software, thus, an executable variant of our product line. With static analyses, we can only analyze static properties, such as structure and data flow. Because the regarded patterns resemble fairly specific class and object structures, we focus on developing an approach that is capable of capturing such structures and detecting them in software systems. According to Rasool et al., existing structural analyses are reasonably successful detecting design patterns that resemble fairly specific structures [29]. Therefore, we reviewed existing static analyses for design pattern detection in order to develop our approach based on existing, reliable techniques.

In our former approach [37], we employed the aforementioned static design pattern detection technique developed by Heuzeroth et al. [20], where they traverse the AST and incrementally check static conditions that have to hold in order to detect patterns. We chose this approach because it was simple to adapt to to our system representation and simple to extend by new design patterns while delivering feasible results. We adapted this approach on feature-oriented software families using the Fuji AST, however, because the Fuji AST lacks necessary information for simple analysis, this approach was only successful to some extent, since we detected plenty of false positives. Nevertheless, the results were promising and, given the fact that our new AST contains all the necessary information and we gain various tool support, we choose to apply this approach once again, however, in an adapted form, combining it with another approach.

As we already mentioned, an AST is basically a graph structure. Therefore, graph-based approaches piqued our interest. Especially the approach of Tsantalis et al. [40], where they use graph pattern matching, combined with a similarity scoring algorithm to identify specific and similar graph structures that constitute design patterns. This approach, especially concerning selected variants of design patterns, seems most suitable for our problem. Therefore, we try to combine both, graph-based analysis and incremental AST traversal in order to achieve a simple yet powerful static detection technique for design patterns.

Nevertheless, we have to compromise on the extent of our approach. According to Heuzeroth et al. [20], they were faced with a number of false positives (i.e., matches that are not design patterns) during their static analysis, which they decreased by subsequent dynamic analysis. Since we cannot make use of dynamic analysis, we have to increase the efficiency of our static analysis by as much as possible. We can only do this by being more specific in the descriptions of the design patterns by focusing on exact representations rather than also trying to detect all variations of a pattern. Because we focus on patterns resembling fairly specific structures, we argue, that we can focus on definite structural descriptions of these patterns.

In addition, Heuzeroth et al. [20] apply lightweight data and control flow analyses in order to be more precise in their findings. For example, they check whether a method calls another specific method within its body. As we need to be precise with the pattern definitions, we also apply such data and control flow analyses as a postprocessing step to structural matches.

5.3.1 Workflow

Our approach combines graph-based detection with incrementally checking conditions on the AST including lightweight data and control flow analyses. The workflow consists of the steps that we illustrate in Figure 5.1.

Describing Structural Conditions

The first step is to describe the necessary class and object collaborations of a design pattern. To this end, we decompose the design pattern by means of object and class collaborations and describe these using role modeling. We already mentioned the use of role modeling as a means to describe design patterns in Section 2.3.2, where we depict an exemplary role model of the strategy pattern in Figure 2.14c. Based on this role model we can define the conditions that have to be met for a given structure to constitute the desired pattern while considering that each role can be played by a different feature-oriented role. Such conditions include, for instance, that a feature-oriented role has to contain a method, or that two feature-oriented roles must or must not contribute to the

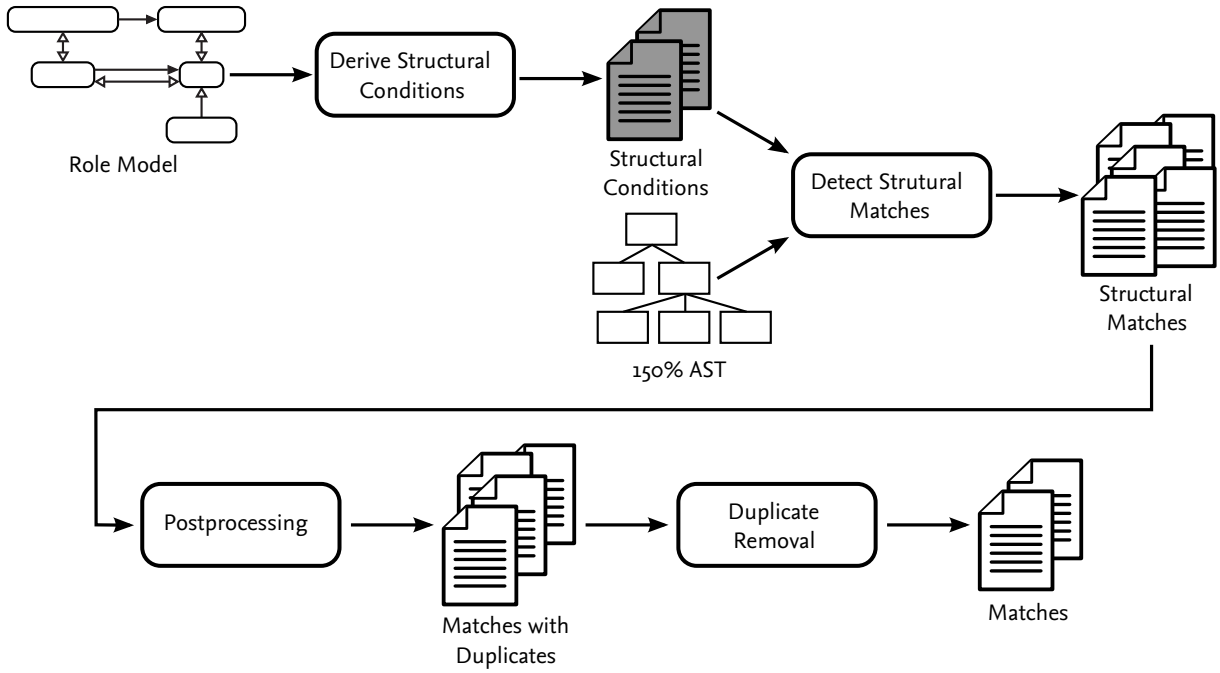


Figure 5.1: General Workflow of our family-based design pattern detection approach

same class. However, based on the approach of Heuzeroth et al. [20], we only describe the specific characteristics of design patterns that describe the minimal structural requirements that have to be met for a given structure to resemble a design pattern.

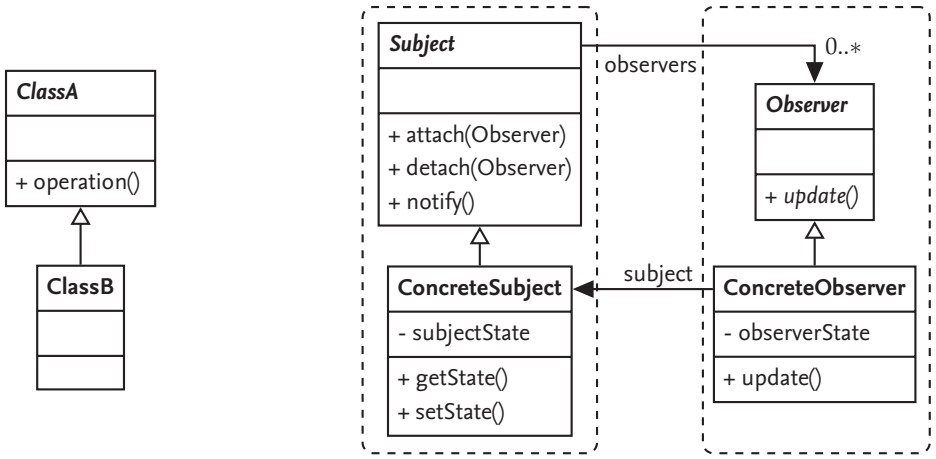
Defining these conditions, we might be able to reuse certain parts of the specification of a pattern for the detection of another, more complex pattern. As we described in Section 2.1.2, several design patterns *use* other design patterns within their solutions, which means, their problem definitions *contain* the problem definition of another pattern. Hence, it might be possible to reuse structural conditions from a design pattern that is used by the currently regarded pattern.

Detecting Structural Matches

The second step is to run the structural detection process on the 150% AST of the selected SPL in order to detect structural matches for the defined conditions. To this end, we describe the structural conditions as graph patterns and apply a pattern matching process to the 150% AST. In Figure 5.2, we depict an example for graph pattern matching on class diagrams. Here, we match a pattern consisting of an inheritance relation of two classes, which is part of almost every design pattern, to the exemplary structure of an observer design pattern instance. First, we have to define a partial graph that describes the structure that we want to discover by describing the desired elements and their relations, thus, the structural conditions that we want to identify. We depict the following structural conditions in Figure 5.2a:

- An abstract class or interface contains a method, respectively.
- A concrete class inherits from or implements the abstract class/interface.

The pattern matching process consists of iterating over each combination of elements in the target graph and checking the defined conditions, resulting in a set of structural matches. In Fig-



(a) Pattern to match (inheritance)

(b) Matches in observer pattern

Figure 5.2: Minimal example for graph pattern matching on the observer pattern

ure 5.2b, we illustrate the matches for this inheritance pattern in the observer design pattern. On the right hand side, the inheritance between the `Observer`, containing the `update` method and the `ConcreteObserver` matches our description. On the left hand side, the `Subject` and `ConcreteSubject` classes match the inheritance condition, however, there are three methods in the `Subject` class that match our pattern, `attach`, `detach` and `notify`. Thus, on the left hand side, there are three matches for our pattern, one for each method of the `Subject` class.

Since, for this step, we can one describe structural conditions when describing design patterns, these matches might contain plenty of false positives. Structural descriptions of design patterns cannot be definite enough to only detect the desired design pattern instances because design patterns are general descriptions that can manifest in various structures (cf. Section 2.1).

Postprocessing

In order to eliminate false positives from the structural matches, we need to conduct a postprocessing step. In our case, this step consists of lightweight data and control flow analyses that describe the design pattern more detailed than the structural conditions. For instance, we check whether a parameter is potentially stored by a specific method (e.g., a possible `addObserver` operation of an observer pattern), or whether a method calls another method within its body (e.g., a template method calls a hook method in a template method pattern). Of course we cannot reliably eliminate all possible false positives, however, the number of remaining false positives can be eliminated with a modest amount of manual work.

Removing Duplicates

The last step, removing duplicate matches, is necessary because design patterns can manifest in various structures, such as an observer pattern with only one or with several concrete observer classes. Therefore, we have to detect the minimal manifestation, which leads to different matches for the different concrete observers. During structural detection, we cannot combine these duplicate matches, but rather have to conduct this step afterwards by combining matches that are manifestations of the same pattern instance, especially regarding multiple concrete classes of the design pattern.

5.3.2 Limitations

While we aim at developing a suitable and reusable approach that fits our requirements, we argue that we cannot develop a universal approach for design pattern detection. As Rasool et al. express in their survey paper, existing design pattern detection techniques all have their limitations [29]. They only cover specific patterns (e.g., focusing more on structural or behavioral patterns) and there is almost no possibility for a conclusive evaluation because there are not many projects where the use of design patterns is documented. This makes the measurement of true negatives (i.e., not detected design patterns) almost impossible.

According to Rasool et al., most approaches apply structural or behavioral analysis, or both, in order to detect certain design patterns [29]. However, a number of patterns resemble similar structural and behavioral properties while only the intent differs. To detect differences in the intent, a semantic analysis would be necessary. And even then, correctly identifying and distinguishing between similar patterns, such as the objectifier and the strategy patterns, might hardly be possible [29].

In our approach, we apply static analyses, which are reliable techniques for detecting patterns resembling fairly specific structures. However, static analyses lack the ability to detect design patterns, whose structural properties are not that definite. Since every design pattern is only a general description of a solution, provided with an exemplary structure, there can be variations of each pattern not exactly resembling this structure. Such design pattern instances are troublesome for design pattern detection, especially for static approaches. Moreover, some design patterns might offer a specific description regarding the problem, however, there is no unique structure that resembles the pattern. This means, that by structurally detecting such patterns, we would get imprecise results containing plenty of false positives. An example for such a pattern is the behavioral pattern *command*, that is used by encapsulating requests as objects and parameterize clients with different requests [16, p. 233 ff.].

Consequently, by means of static analyses, we have to find a trade-off between detecting design patterns as generally as possible to include as many variations as possible, and detecting design patterns as precisely as possible, to exclude as many false positives as possible. The first approach would increase the number of false positives whereas the latter approach would increase the number of true negatives. As we described before, in our approach we focus on a precise detection, which means, for static analysis, describing a fairly precise structure of the design pattern combined with data and control flow analyses to be even more accurate. Consequently, we disregard pattern variations, as a more general description would increase the number of false positives and make our approach inefficient.

5.4 Implementation

In order to run a graph-based analysis for a design pattern, we need to describe the pattern representation in a graph-based way. With EMF offering plenty of tool support, we selected EMF-IncQuery[1], a tool for high performance graph searches on EMF models. EMF-IncQuery provides us with a powerful declarative query language, which we can use to describe the structural conditions of design patterns. We implemented our approach resulting in the tool called Pattern Demon, a plugin for Eclipse that is integrated with FOPJaMoPP. We base Pattern Demon on the graph analysis

[1] EMF-IncQuery: http://www.eclipse.org/incquery/

```
1  package queries.subtype
2  import "http://www.emftext.org/java/classifiers"
3  import "http://www.emftext.org/java/types"
4  import "http://www.tu-braunschweig.de/isf/fopjava/types"
5
6  pattern transitiveSubClass(superClass:Class, subClass:Class) {
7    find isSubClass+(superClass, subClass);
8  }
9  pattern isSubClass(superClass:Class, subClass:Class) {
10   Class.^extends(subClass, extendsRef);
11   NamespaceClassifierReference.classifierReferences(extendsRef, extendsClassRef);
12   FopClassifierReference.targets(extendsClassRef, superClass);
13 } or {
14   Class.^extends(subClass, extendsRef);
15   FopClassifierReference.targets(extendsRef, superClass);
16 }
```

Figure 5.3: EMF-IncQuery query language example – Matching all tuples of classes that resemble a transitive extends relation

tool EMF-IncQuery, which we use in version 0.8.1. In the following, we give a brief introduction to EMF-IncQuery, after which we describe the architecture of Pattern Demon.

5.4.1 EMF-IncQuery

EMF-IncQuery features high performance graph search on EMF models using declarative queries that can be executed automatically. In order to define such queries, EMF-IncQuery provides a powerful query language. We can use the query language to describe conditions on our EMF model in order to find instances of the described structures. Using the API of EMF-IncQuery, we are able to run the pattern detection automatically on an EMF model using the specified queries. EMF-IncQuery automatically checks the queries incrementally and efficiently and returns a set of matches.

In Figure 5.3, we depict an example for the query language, where we apply EMF-IncQuery to our 150% AST and match all tuples of classes that resemble a transitive extends relation. With this query language, we can express all kinds of relations between elements of an EMF model.

The pattern definition starts with a package declaration, after which we need to define with imports, which EMF packages we are employing. Starting with the `pattern` keyword, we can describe a pattern definition using queries by first defining the parameters. With the parameters, we define, for which elements EMF-IncQuery checks all possible combinations, thus, which elements EMF-IncQuery tries to match in the EMF model. We can describe multiple pattern definitions and reuse them within other pattern definitions.

For each pattern, we can now define various queries that have to evaluate to true in order for a combination of parameter objects to be considered a match. With a query, we can specify a certain relationship between two elements. For instance, in Line 10, we express that the element called `subClass` of type `Class` has to have an extends relation with another element that we call `extendsRef` because the `extends` field of class contains an element of type `ClassifierReference` or `NamespaceClassifierReference`. We can now use the element `extendsRef` to define more de-

tailed queries. Because different relations can be possible, such as different types of objects in the `extends` field of `Class`, we can use the `or` keyword to define multiple possibilities of queries for a certain pattern. Because `extends` is also a keyword in the query language, we have to use the circumflex (^) to use it as an identifier.

Using the `find` keyword, we can call other pattern definitions, such as we do in Line 7. Here, we call the pattern definition of `isSubClass` for our two parameters. With the plus (+) symbol, we can specify a transitive relation, thus, in addition to a direct subclass, subclass of a subclass is also identified with this query.

After we specified the queries for the graph pattern we want to match, EMF-IncQuery uses a code generator to generate a `Matcher` class for each pattern description. In our example, we get two `Matchers`, the `TransitiveSubClassMatcher` and the `IsSubClassMatcher`. Moreover, EMF-IncQuery generates `Match` classes for each pattern, such that we get a `TransitiveSubClassMatch` and an `IsSubClassMatch`. For each parameter of the pattern description, the corresponding `Match` class contains a field where the specific objects of a match can be stored. Using the API of EMF-IncQuery, we can execute a matcher on an existing EMF model resulting in a set of `Match` objects.

If we now call the `transitiveSubClass` pattern on an AST using the EMF-IncQuery API, EMF-IncQuery checks all combinations of two classes that exist within the AST and incrementally evaluates all queries for each combination. Afterwards, a `Match` object for each match is returned, which can now be further analyzed or used.

5.4.2 Architecture

As we already mentioned, we also realized Pattern Demon as an Eclipse plugin. It contains the following plugin projects:

`de.tu_bs.cs.isf.patterndemon`

The root project contains common parts of the detection process. The abstract class `Detector` contains a template method regarding the initialization of EMF-IncQuery, the start of the detection process and printing of the results. Moreover, it contains common operations to check specific conditions, for example, data flow analyses that have to be performed for several design patterns, such as checking whether a method calls another specific method in its body. Furthermore, it contains operations to analyze the results more closely, for instance, an operation that computes the containing feature for each element of a match using the feature annotation of each compilation unit (cf. Section 4.4.4).

Because, after the postprocessing and especially after the duplicate removal, matches might contain more elements than the generated `Match` objects of EMF-IncQuery can store, we also provide a `FinalMatch` interface that can be used to store matches after postprocessing.

`de.tu_bs.cs.isf.patterndemon.patterns`

This project contains one package for each design pattern comprising a pattern specification in the EMF-IncQuery query language, a concrete `Detector` implementation, as well as a concrete implementation of the `FinalMatch` interface to store the matches after postprocessing. The concrete `Detector` implementation runs the detection process by executing the necessary steps to run the `Matcher` class generated by EMF-IncQuery. Moreover, it contains the necessary postprocessing including duplicate removal and a `print` operation for the results.

de.tu_bs.cs.isf.patterndemon.ui
: This project contains the extensions to the Eclipse user interface. For each concrete `Detector` class and, thus, for each design pattern, we extend the user interface with a menu item to start the detection process.

This way, Pattern Demon is implemented highly extensibly. For a new design pattern, a developer only has to introduce the design pattern specification using the EMF-IncQuery query language, a concrete `Detector` class that contains the necessary steps to run the `Matcher` class generated by EMF-IncQuery, as well as the required postprocessing and printing of the corresponding `FinalMatches`. Also, a `FinalMatch` class has to be provided, containing the objects that, together, constitute a unique match. Moreover, the Eclipse user interface has to be extended with a menu item to trigger the detection process.

5.5 Detecting Design Patterns with Pattern Demon

In this section, we illustrate the pattern detection process using Pattern Demon on the four patterns *template method*, *observer*, *strategy* and *composite*.

5.5.1 Common Pattern Rules

In order to reuse pattern rules across the structural conditions of different design patterns, we created a collection of smaller, reusable graph patterns for recurring constraints. In the following, we will give a brief overview on these recurring sets of queries.

In Figure 5.5, we depict patterns to identify the supertype relations between classes as well as between classes and interfaces.

- The pattern `rolesContributeToSameClass` in Line 1 checks whether two specific feature-oriented roles contribute to the same class, thus, whether their compilation units share the same qualified name.
- The pattern `multipleChildRoles` in Line 8 checks whether a specific role has multiple child roles of any kind (implementing or inheriting).
- The pattern `implementsRole` in Line 18 checks whether a specific interface-role is implemented by a class-role.
- The pattern `extendsRole` in Line 27 checks whether a specific role extends another role.
- The pattern `childRole` in Line 36 checks whether two roles have a parent-child relation of any kind, however, they must not be the same role.
- The pattern `sameOrChildRole` in Line 42 checks whether two roles are the same or have a parent-child relation of any kind.

In Figure 5.6, we depict two patterns concerning fields. In particular, we check whether a role holds a field of a specific kind.

- In Line 1, we check whether a role holds a field of a specific type.
- In Line 8, we check whether a role holds an array of a specific type.

```
1  pattern implementsMethod(method:Method, childRole:Class) {
2    Class.members(childRole, subRoleMethod);
3    Method.name(subRoleMethod, methodName);
4    Method.name(method, methodName);
5  }
6
7  pattern methodHasParameterOfType(
8    method:Method, parameter:Parameter, type:ConcreteClassifier) {
9    Method.parameters(method, parameter);
10   Parameter.typeReference(parameter, paramRef);
11   NamespaceClassifierReference.classifierReferences(paramRef, paramRoleRef);
12   FopClassifierReference.targets(paramRoleRef, paramType);
13   find rolesContributeToSameClass(paramType, type);
14 }
```

Figure 5.4: EMF-IncQuery patterns: Methods and parameters

- In Line 22, we check whether a role holds a field containing a specific type argument.
- In Line 42, we check whether a role holds a field of any previously described kind.

In Figure 5.4, we depict two patterns concerning methods.

- `implementsMethod`, in Line 1 checks whether a specific role implements a specific method. However, we only check whether we can find a method with the same name as the desired one, disregarding the complete signature (i.e., the parameters) for now. Because even if the method has the same parameters, these can be parsed as different feature-oriented roles contributing to the same class (cf. Section 4.4.4). Hence, matching the parameters is a difficult task that we disregard for now, accepting that false positive results might occur at this point.
- `methodHasParameterOfType`, in Line 7, checks whether a method signature contains a parameter of a specific type by checking whether the parameter type (i.e., the feature-oriented role that is parsed as the parameter) contributes to the same class as the regarded feature-oriented role.

```
1  pattern rolesContributeToSameClass(role1, role2) {
2    CompilationUnit.name(cu1, cuName);
3    CompilationUnit.name(cu2, cuName);
4    CompilationUnit.classifiers(cu1, role1);
5    CompilationUnit.classifiers(cu2, role2);
6  }
7
8  pattern multipleChildRoles(
9    parentRole:ConcreteClassifier,
10   childRole1:Class,
11   childRole2:Class
12 ) {
13   find childRole+(parentRole, childRole1);
14   find childRole+(parentRole, childRole2);
15   childRole1 != childRole2;
16 }
17
18 pattern implementsRole(parentRole:ConcreteClassifier, childRole:Class) {
19   Class.implements(childRole, implRef);
20   NamespaceClassifierReference.classifierReferences(implRef, implItfRef);
21   FopClassifierReference.targets(implItfRef, parentRole);
22 } or {
23   Class.implements(childRole, implRef);
24   FopClassifierReference.targets(implRef, parentRole);
25 }
26
27 pattern extendsRole(parentRole:ConcreteClassifier, childRole:Class) {
28   Class.^extends(childRole, extendsRef);
29   NamespaceClassifierReference.classifierReferences(extendsRef, extendsClassRef);
30   FopClassifierReference.targets(extendsClassRef, parentRole);
31 } or {
32   Class.^extends(childRole, extendsRef);
33   FopClassifierReference.targets(extendsRef, parentRole);
34 }
35
36 pattern childRole(parentRole:ConcreteClassifier, childRole:Class) {
37   find extendsRole+(parentRole, childRole);
38 } or {
39   find implementsRole+(parentRole, childRole);
40 }
41
42 pattern sameOrChildRole(parentRole:ConcreteClassifier, childRole:Class) {
43   parentRole == childRole;
44 } or {
45   find childRole(parentRole, childRole);
46 }
```

Figure 5.5: EMF-IncQuery patterns: Checking relations between feature-oriented roles

```
1  pattern holdsFieldOfType(holder:Class, type:ConcreteClassifier) {
2    Class.members(holder, field);
3    Field.typeReference(field, fieldTypeRef);
4    NamespaceClassifierReference.classifierReferences(fieldTypeRef, fieldClassRef);
5    FopClassifierReference.targets(fieldClassRef, type);
6  }
7
8  pattern holdsArrayOfType(holder:Class, type:ConcreteClassifier) {
9    Class.members(holder, field);
10   Field.typeReference(field, fieldTypeRef);
11   Field.arrayDimensionsBefore(field, _arrayDimBefore);
12   NamespaceClassifierReference.classifierReferences(fieldTypeRef, fieldClassRef);
13   FopClassifierReference.targets(fieldClassRef, type);
14 } or {
15   Class.members(holder, field);
16   Field.typeReference(field, fieldTypeRef);
17   Field.arrayDimensionsAfter(field, _arrayDimAfter);
18   NamespaceClassifierReference.classifierReferences(fieldTypeRef, fieldClassRef);
19   FopClassifierReference.targets(fieldClassRef, type);
20 }
21
22 pattern holdsFieldWithTypearg(holder:Class, typearg:ConcreteClassifier) {
23   Class.members(holder, field);
24   Field.typeReference(field, fieldTypeRef);
25   NamespaceClassifierReference.classifierReferences(fieldTypeRef, fieldClassRef);
26   FopClassifierReference.typeArguments(fieldClassRef, fieldTypeArg);
27   QualifiedTypeArgument.typeReference(fieldTypeArg, fieldTypeArgTypeRef);
28   NamespaceClassifierReference.classifierReferences(
29     fieldTypeArgTypeRef, fieldTypeArgClassRef);
30   FopClassifierReference.targets(fieldTypeArgClassRef, typearg);
31 } or {
32   Class.members(holder, field);
33   Field.typeReference(field, fieldTypeRef);
34   NamespaceClassifierReference.classifierReferences(fieldTypeRef, fieldClassRef);
35   FopClassifierReference.typeArguments(fieldClassRef, fieldTypeArg);
36   ExtendsTypeArgument.extendTypes(fieldTypeArg, fieldTypeArgTypeRef);
37   NamespaceClassifierReference.classifierReferences(
38     fieldTypeArgTypeRef, fieldTypeArgClassRef);
39   FopClassifierReference.targets(fieldTypeArgClassRef, typearg);
40 }
41
42 pattern holdsFieldOrListOrArrayOfType(holder:Class, type:ConcreteClassifier) {
43   find holdsFieldOfType(holder, type);
44 } or {
45   find holdsArrayOfType(holder, type);
46 } or {
47   find holdsFieldWithTypearg(holder, type);
48 }
```

Figure 5.6: EMF-IncQuery patterns: Holding fields, arrays and lists

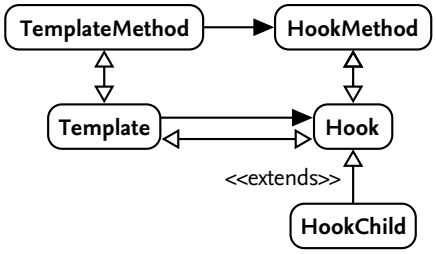


Figure 5.7: Role model of the template method pattern

5.5.2 Template Method

The template method pattern (cf. Section 2.1.3) is a simple behavioral pattern, which is used to specify a skeleton for an algorithm and define concrete steps in subclasses [16, p. 325 ff.]. We view the template method pattern as two classes, an abstract class, introducing `templateMethod` and at least one `hookMethod`, which is called from the `templateMethod`, and its concrete child class, defining the `hookMethod`.

Pattern Specification

The first step for the detection of a pattern consists of specifying the conditions that must be met in order for a graph structure to resemble the pattern. In Figure 2.1, we illustrate the exemplary structure of a template method patterns by Gamma et al. [16]. Moreover, in Figure 5.7, we depict the corresponding role model for the template method pattern. Based on these roles, we can now describe the structural conditions that have to be met while considering that each role can be played by a different feature-oriented role.

We define the structural conditions of the template method pattern using the query language of EMF-IncQuery in Figure 5.8. As we described in Section 5.4.1, we first have to specify the package and afterwards import all required EMF packages. In Line 1, we start the pattern description of the template method.

In Figure 5.7, we show that a template method pattern consists of five roles that we define as parameters for the pattern:

`template`
: The (abstract) template role containing the template method, thus, the skeleton of the algorithm.

`templateMethod`
: The template method containing the skeleton of the algorithm.

`hook`
: The class containing the declaration of the (abstract) hook method that is defined in at least one child class.

`hookMethod`
: The hook method declaration that is contained by the hook role.

`hookChild`
: A concrete child class of the hook role that implements the hook method.

```
1  pattern TemplateMethod (
2    template:Class, templateMethod:Method,
3    hook:Class, hookMethod:Method, hookChild:Class
4  ) {
5    CompilationUnit.name(templateCu, cuName);
6    CompilationUnit.name(hookCu, cuName);
7    CompilationUnit.classifiers(templateCu, template);
8    CompilationUnit.classifiers(hookCu, hook);
9
10   Class.members(template, templateMethod);
11   Class.members(hook, hookMethod);
12   templateMethod != hookMethod;
13
14   find isChildRole+(hook, hookChild);
15   find roleImplementsMethod(hookMethod, hookChild);
16 }
```

Figure 5.8: EMF-IncQuery pattern specification for the template method pattern

The following structural conditions have to be met in order for these five elements to resemble the template method pattern:

1. The `template` role and `hook` role must contribute to the same class, which means that their `CompilationUnits` share the same qualified name. They can also be the same feature-oriented role.
2. The `templateMethod` must be contained by the `template` role, the `hookMethod` must be contained by the `hook` role.
3. The `templateMethod` and the `hookMethod` must not be the same method.
4. There has to be at least one feature-oriented role inheriting from the `hook` role and implementing the `hookMethod`.

We implement these conditions as follows. The first condition, whether `template` role and `hook` role contribute to the same class, is checked starting in Line 5. In Line 5 and Line 6, we check for two compilation units `templateCu` and `hookCu` sharing the same name `cuName`. Afterwards, in Line 7 and Line 8, we check, whether these two compilation units contain the `templateRole` and `hookRole`, respectively.

The second condition is checked in Line 10 and Line 11, where we check, whether the `template` role and `hook` role contain the `templateMethod` and `hookMethod`, respectively.

In Line 12, we check the third condition, whether `templateMethod` and `hookMethod` are not the same method declaration.

Starting in Line 14, we check the fourth condition, whether at least one feature-oriented role inheriting from `hook` role exists that implements the `hookMethod`. To this end, we use two common pattern definitions `childRole` in Figure 5.5 and `implementsMethod` in Figure 5.4, which check, whether an extends relation between two feature-oriented roles exist and whether a method is implemented within a feature-oriented role, respectively. We call the `childRole` pattern in Line 14, where we also

consider transitive relations of inheritance. In Line 15, we call the `implementsMethod` pattern to check whether the feature-oriented role inheriting from `hook` role implements the `hookMethod`.

Postprocessing

From the role model of the template method pattern in Figure 5.7, we can derive one more condition that has to be checked:

5. The body of the `templateMethod` must contain a call to the `hookMethod`.

We cannot efficiently check this fifth condition using the query language because a method call can be contained in any kind of nested expression. We would have to check every individual case of how a method call can be contained within a method body (which is a similar problem to exchanging every element reference in the AST to create the 150% AST as we described in Section 4.4.4).

However, we can check such data and control flow conditions using postprocessing steps on the `Match` objects that are created after executing the `Matcher` of EMF-IncQuery. In this case, we only have to iterate through the contents of the `templateMethod`, and if we discover a method call, check whether this method call targets the `hookMethod`. This is a simple operation done in Java, whereas the query specification would have been endless.

Removing Duplicates

The last step is to remove duplicates from the matches. In the case of the template method pattern, the unique part of a match consists of the `template` and the `templateMethod`. Only if these elements are the same for two matches, we assume, that these matches are duplicates. Each new combination of `template` and `templateMethod`, we consider to be a new instance of the pattern. In contrast, there can be more than one `hook`, `hookMethod` or `hookChild` within one instance of the template method pattern. Thus, if we discover duplicates, we combine them to one `FinalMatch` object containing all the combinations of `hook`, `hookMethod` and `hookChild` that are detected for one combination of `template` role and `templateMethod`.

After the duplicate removal, we get a collection of `FinalMatch` objects containing all detected instances of template method patterns. For now, as a last step, this collection of matches is printed to the console using the print operation that is specified within each `Detector`. However, further processing or analysis on these matches is possible.

5.5.3 Observer

The observer pattern (cf. Section 2.1.3) is a behavioral design pattern, which is used to to notify dependent objects automatically, if the object they depend on changes its state, while keeping a

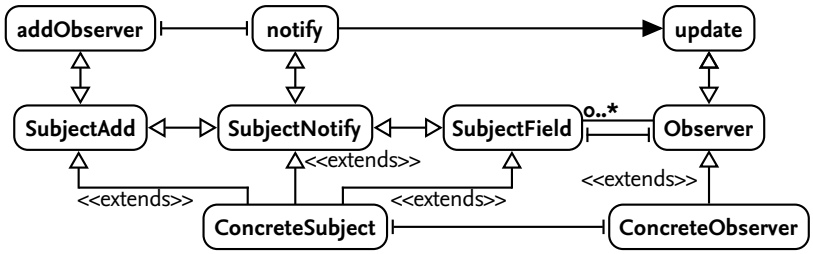


Figure 5.9: Role model of the observer pattern

```
1  pattern Observer(
2    subjectAdd:ConcreteClassifier, addMethod:Method, addMethodParameter:Parameter,
3    subjectNotify:ConcreteClassifier, notifyMethod:Method,
4    subjectField:ConcreteClassifier,
5    observer:ConcreteClassifier, updateMethod:Method,
6    concreteSubject:Class, concreteObserver:Class
7  ) {
8    find rolesContributeToSameClass(subjectAdd, subjectNotify); // 1
9    find rolesContributeToSameClass(subjectAdd, subjectField); // 1
10
11   find holdsFieldOrListOrArrayOfType(subjectField, observer); // 2
12   subjectAdd != observer; // 1
13   subjectNotify != observer; // 1
14   subjectField != observer; // 1
15
16   ConcreteClassifier.members(subjectAdd, addMethod); // 3
17   ConcreteClassifier.members(subjectNotify, notifyMethod); // 3
18   addMethod != notifyMethod; // 3
19
20   find methodHasParameterOfType(addMethod, addMethodParameter, observer); // 4
21
22   ConcreteClassifier.members(observer, updateMethod); // 5
23
24   find sameOrChildRole+(subjectAdd, concreteSubject); // 6
25   find sameOrChildRole+(observer, concreteObserver); // 7
26   concreteSubject != concreteObserver; // 8
27
28   find roleImplementsMethod(updateMethod, concreteObserver); // 9
29 }
```

Figure 5.10: EMF-IncQuery pattern specification for the observer pattern

loosely coupled system [16, p. 293 ff.]. Based on Heuzeroth et al. [20], we view the observer pattern as two classes, a `Subject` and an `Observer`. The `Subject` class holds an `addObserver` and a `notify` method, the `Observer` class holds an `update` method. The `addObserver` method stores its parameter, while the `notify` method calls the `update` method within its body. `Subject` and `Observer` must not be the same class, `ConcreteSubject` and `ConcreteObserver` are optional classes inheriting from `Subject` and `Observer`, respectively [20].

Pattern Specification

In Figure 5.9, we illustrate the role model for the family-based detection of an instance of the observer pattern. In particular, we have six roles played by feature-oriented roles:

`SubjectAdd`
: Played by the feature-oriented role introducing the `addObserver` method to the `Subject` class. This feature-oriented role must contribute to the same `Subject` class as the feature-oriented roles playing the `SubjectNotify` and `SubjectField` roles, but must not contribute to the same class as the feature-oriented role playing the `Observer` role.

SubjectNotify
Played by the feature-oriented role introducing the `notify` method to the `Subject` class. This feature-oriented role must contribute to the same `Subject` class as the feature-oriented roles playing the `SubjectAdd` and `SubjectField` roles, but must not contribute to the same class as the feature-oriented role playing the `Observer` role.

SubjectField
Played by the feature-oriented role holding a reference to the `Observer` class. This feature-oriented role must contribute to the same `Subject` class as the feature-oriented roles playing the `SubjectAdd` and `SubjectNotify` roles, but must not contribute to the same class as the feature-oriented role playing the `Observer` role.

Observer
Played by the feature-oriented role introducing the `update` method to the `Observer` class.

ConcreteSubject
Played by the feature-oriented role introducing the `ConcreteSubject` class that inherits from the `Subject` class. This role can also be played by the feature-oriented role playing the `SubjectAdd` or `SubjectNotify` role. This role must not be played by the feature-oriented role playing the `ConcreteObserver` role.

ConcreteObserver
Played by the feature-oriented role introducing the `ConcreteObserver` class that inherits from the `Observer` class. This role can also be played by the feature-oriented role playing the `Observer` role. This role must not be played by the feature-oriented role playing the `ConcreteSubject` role.

Moreover, we have three roles played by methods:

addObserver
The method introduced by the feature-oriented role playing the `SubjectAdd` role, which must not be the same method as the method playing the `notify` role.

notify
The method introduced by the feature-oriented role playing the `SubjectNotify` role, which must not be the same method as the method playing the `addObserver` role.

update
The method introduced by the feature-oriented role playing the `Observer` role.

The following structural conditions have to be met in order for a given structure to constitute an observer pattern:

1. The feature-oriented roles playing the `SubjectAdd`, `SubjectNotify` and `SubjectField` roles must contribute to the same `Subject` class. None of the feature-oriented roles playing one of the `Subject` roles may be playing the `Observer` role.
2. The `SubjectField` must hold a reference to the `Observer` class, thus, have a field (array, list) with the type (or type argument) of the `Observer` class.

3. The `addObserver` method must be contained by the `SubjectAdd` role, the `notify` method must be contained by the `SubjectNotify` role. They must not be the same method.
4. The addObserver method must have the `Observer` as a parameter.
5. The `update` method must be contained by the `Observer` role.
6. The `ConcreteSubject` role must be played by a feature-oriented role inheriting from the `Subject` class or by a feature-oriented role contributing to the `Subject` class.
7. The `ConcreteObserver` role must be played by a feature-oriented role inheriting from the `Observer` class or by a feature-oriented role contributing to the `Observer` class.
8. The `ConcreteSubject` and `ConcreteObserver` roles must not be played by feature-oriented roles contributing to the same class.
9. The `ConcreteObserver` has to define the `update` method.

In Figure 5.10, we depict the EMF-IncQuery pattern specification for the observer pattern, checking the listed structural conditions. Here, we incrementally check the aforementioned conditions, mostly using common pattern definitions that we explained in Section 5.5.1. Using comments, we mark which query contributes to checking which structural condition.

Postprocessing

From the role model in Figure 5.9 and the specification of the observer pattern [16, p. 293 ff.], we can derive the following necessary data and control flow checks:

10. The `notify` method must contain a call to the `update` method.
11. The `addObserver` method must somehow store its parameter.

In Section 5.5.2, we already explain checking the tenth condition. For the eleventh condition, we need a proper definition of what requirements a method has to fulfill in order to be specified as a potential *store* method for its parameter. According to Heuzeroth et al. [20], it is sufficient to check the following conditions in order to rule out most false positives:

- The parameter is used on the right hand side of an assignment (potential store assignment)
- The parameter is passed to another method (potential store method, e.g., `add` method of `java.util.List`)

These conditions are easily checked by iterating through all expressions of the method body and searching for assignments and method calls, where the parameter is used within the expression of the assignment operation or within the argument of the method call, respectively.

Removing Duplicates

We only check for the presence of one feature-oriented role playing the `ConcreteSubject` and `ConcreteObserver` roles, respectively, which can also be the same feature-oriented roles playing one of the `Subject` or the `Observer` roles, respectively. Hence, if for two detect pattern instances, only the `ConcreteSubject` or `ConcreteObserver` roles are played by different feature-oriented roles, we treat these as duplicates, which we combine to one pattern instance.

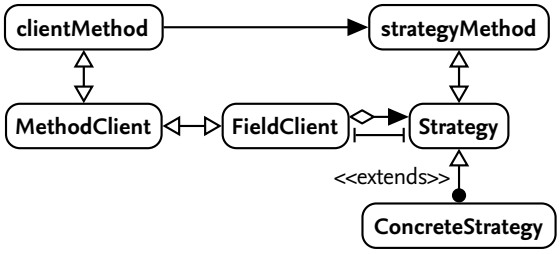


Figure 5.11: Role model of the strategy pattern

5.5.4 Strategy/Objectifier

The strategy pattern is applied specifically to objectify similar, yet different, interchangeable algorithms [16, p. 315 ff.]. It is a specialization of the objectifier pattern [42], which is why they are structurally equal (cf. Section 2.1.3). Even though we use the term "strategy" in this section, with this detection technique we regard both the strategy and objectifier patterns.

We view the strategy/objectifier pattern as a class `Client`, holding a reference to a class `Strategy`, both being different classes. The `clientMethod`, contained by the `Client`, calls the `strategyMethod`, contained by the `Strategy`, within its body. Moreover, multiple classes inheriting from the `Strategy` class and implementing the `strategyMethod` exist.

Pattern Specification

In Figure 5.11, we illustrate the role model for the family-based detection of an instance of the strategy pattern. We cannot detect an arbitrary number of concrete strategies using EMF-IncQuery, but rather can only check whether multiple concrete strategies exist by checking for tuples of concrete strategies. In particular, we have five roles played by feature-oriented roles:

`MethodClient`
: Played by the feature-oriented role contributing to the same class as `FieldClient` and containing the `clientMethod`.

`FieldClient`
: Played by the feature-oriented role contributing to the same class as `MethodClient` and holding a reference to the `Strategy` class.

`Strategy`
: Played by the feature-oriented role containing the `strategyMethod`.

`ConcreteStrategy1`
: Since multiple child classes of the `Strategy` class are required, we need at least two concrete strategies. The feature-oriented role playing the `ConcreteStrategy1` role must be contributing to a different class as the feature-oriented role playing the `ConcreteStrategy2` role.

`ConcreteStrategy2`
: The second concrete strategy is introduced by a feature-oriented role playing the `ConcreteStrategy2` role, contributing to a different class as the feature-oriented role playing the `ConcreteStrategy1` role.

```
1  pattern Strategy (
2    methodClient:Class, clientMethod:Method, fieldClient:Class,
3    strategy:ConcreteClassifier, strategyMethod:Method,
4    concreteStrategy1:Class, concreteStrategy2:Class
5  ) {
6    find rolesContributeToSameClass(methodClient, fieldClient); // 1
7    neg find rolesContributeToSameClass(methodClient, strategy); // 2
8
9    find holdsFieldOfType(fieldClient, strategy); // 3
10
11   Class.members(methodClient, clientMethod); // 4
12   ConcreteClassifier.members(strategy, strategyMethod); // 4
13
14   find multipleChildRoles(strategy, concreteStrategy1, concreteStrategy2); // 5
15   find roleImplementsMethod(strategyMethod, concreteStrategy1); // 5
16   find roleImplementsMethod(strategyMethod, concreteStrategy2); // 5
17 }
```

Figure 5.12: EMF-IncQuery pattern specification for the strategy pattern

Moreover, we have two roles played by methods:

`clientMethod`
: The method contained by the feature-oriented role playing the `Client` role.

`strategyMethod`
: The method contained by the feature-oriented role playing the `Strategy` role.

The following structural conditions have to be met in order for a given structure to constitute a strategy pattern:

1. The feature-oriented roles playing the `MethodClient` and `FieldClient` roles must contribute to the same class.
2. The feature-oriented roles playing the `MethodClient`/`FieldClient` and `Strategy` roles must not contribute to the same class.
3. The feature-oriented roles playing the `FieldClient` role must hold a reference to the `Strategy` class.
4. The `clientMethod` must be contained by the `MethodClient`, the `strategyMethod` must be contained by the `Strategy`.
5. There have to be at least two concrete strategies and, thus, two feature-oriented roles contributing to different classes that are both child classes of the `Strategy`. Both of them have to define the `strategyMethod`.

In Figure 5.12, we depict the EMF-IncQuery pattern specification for the strategy pattern, checking the listed structural conditions. Here, we incrementally check the aforementioned conditions, mostly using common pattern definitions that we explained in Section 5.5.1. Using comments, we mark which query contributes to checking which structural condition.

Postprocessing

From the role model in Figure 5.11, we can derive the following necessary control flow condition:

6. The `clientMethod` must contain a call to the `strategyMethod`.

In Section 5.5.2, we already explain checking this sixth condition.

Removing Duplicates

Since we are searching for pattern instances with exactly two concrete strategies, each pattern instance containing more than two concrete strategies is detected multiple times for each combination of concrete strategies, which we combine to a `FinalMatch` object containing all concrete strategies for each combination of feature-oriented roles detected for playing the `Client` and `Strategy` roles.

5.5.5 Composite

The composite pattern provides the developer with the ability to compose objects in a hierarchical tree-structure to represent part-whole hierarchies [16, p. 163 ff.]. We view the composite pattern as two classes, a `Component` and its child class `Composite`, which holds a reference to its parent class and an `addComponent` method that stores its parameter [20].

Pattern Specification

In Figure 5.13, we illustrate the role model for the family-based detection of an instance of the composite pattern. In particular, we have two roles played by feature-oriented roles:

`Composite`
: The feature-oriented role playing the `Composite` role holds a reference to objects of its own supertype as well as the `addComponent` method.

`Component`
: The supertype of the feature-oriented role playing the `Composite` role, which is used as a parameter for the `addComponent` method.

Moreover, we have one role played by a method:

`addComponent`
: The method that is contained by the feature-oriented role playing the `Composite` role and whose parameter, which is stored within the method body, is of the type of the class the feature-oriented role playing the `Component` role contributes to.

The following structural conditions have to be met in order for a given structure to constitute a composite pattern:

1. The feature-oriented role playing the `Composite` role must be the same or the child type of the feature-oriented role playing the `Component` role.
2. The feature-oriented role playing the `Composite` role must hold a reference to objects of the type that the feature-oriented role playing the `Component` role contributes to.

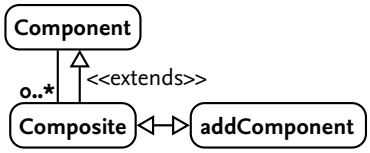


Figure 5.13: Role model of the composite pattern

3. The feature-oriented role playing the `Composite` role must contain the `addComponent` method, which has the `Component` as a parameter.

In Figure 5.14, we depict the EMF-IncQuery pattern specification for the composite pattern, checking the listed structural conditions. Here, we incrementally check the aforementioned conditions, mostly using common pattern definitions that we explained in Section 5.5.1. Using comments, we mark which query contributes to checking which structural condition.

Postprocessing

From the specification of the composite pattern [16, p. 163 ff.], we can derive the following necessary data flow condition:

4. The `addComponent` method must somehow store its parameter.

In Section 5.5.3, we already explain checking this fourth condition.

Removing Duplicates

For the composite pattern, there is no duplicate removal because we cannot reliably specify conditions for two matches to be duplicates.

```
1  pattern Composite (
2    composite:Class, component:ConcreteClassifier,
3    addMethod:Method, addMethodParameter:Parameter,
4  ) {
5    find sameOrChildRole+(component, composite); // 1
6
7    find holdsFieldOrListOrArrayOfType(composite, component); // 2
8
9    Class.members(composite, addMethod); // 3
10   find methodHasParameterOfType(addMethod, addMethodParameter, component); // 3
11 }
```

Figure 5.14: EMF-IncQuery pattern specification for the composite pattern

5.6 Summary

With Pattern Demon, we gain a powerful and extensible tool for family-based design pattern detection on a 150% AST created with FOPJaMoPP. Using EMF-IncQuery, we gain the possibility to describe conditions and relations of feature-oriented roles instead of mere classes with ease. Moreover, the combination of a graph-based structural analysis combined with lightweight data flow

analyses, based on the work of Tsantalis et al. [40] and Heuzeroth et al. [20], offers a way of precisely describing static attributes of design patterns, including simple variations, in order to detect them efficiently. Nevertheless, as we mentioned in Section 5.3.2, Pattern Demon is not universally applicable because several design patterns do not at all resemble a definite structure and are therefore hard to detect using only static analyses. Moreover, we cannot evaluate the accuracy of Pattern Demon regarding true negatives (i.e., missed pattern instances) because there is hardly any documentation on the use of design patterns in software. For the design patterns, for which we implemented the detection, we evaluated Pattern Demon by means of minimal examples for common implementations of the respective design pattern. However, this does not imply that all variations of these patterns can be detected. Nevertheless, for our purposes, Pattern Demon seems highly suitable.

6 Case Study

In the last two sections, we described FOPJaMoPP, the implementation of a variability-aware system representation for feature-oriented Java code, as well as Pattern Demon, an automated, family-based pattern detection technique based on the 150% AST of FOPJaMoPP. During this case study, we employ both approaches in order to analyze our research goals expressed in Section 1.2.

In the following, we first state our research questions necessary to achieve our research goals in Section 6.1. Next, we describe the setup of our case study in Section 6.2 and the methodology in Section 6.3. Afterwards, we depict our results in Section 6.4, leading to a discussion in Section 6.5. We address threats to validity in Section 6.6, before we finish with a brief summary in Section 6.7.

6.1 Research Questions

In prior work [37], we already revealed the existence of decomposed design patterns in feature-oriented SPLs, however, we were unable to derive guidelines concerning their application across features. In order to analyze our research goals of revealing the decomposed application of design patterns and deriving a variability-aware design pattern catalog, we need to answer the following questions:

Research Question 1: *How are design pattern implementations decomposed along features?*

In order to reveal the variability-aware application of design patterns, we need to find out, how design patterns are decomposed along features. Design patterns consist of different collaborations that, when using a modular programming approach, such as FOP, could be implemented in several different modules. Hence, we need to reveal, which collaborations of a design pattern are implemented in which feature-oriented roles of the product line.

Research Question 2: *How are the involved features related to each other?*

As soon as we know that multiple features are involved in implementing a design pattern, we need to reveal the feature relationships in terms of the feature model to derive general application rules for a design pattern. Hence, we need to capture the feature collaborations necessary to implement the design pattern using a family role model (cf. Section 3.3).

Research Question 3: *What variability-aware application is common for the analyzed design patterns?*

As we mentioned in Section 3.2, we aim at creating a variability-aware design pattern catalog including guidelines on the application of design patterns. Using our data on feature collaborations of a decomposed design pattern, we can discuss the implications of decomposing the pattern. Moreover, we examine the structural feature interactions caused by the variability-aware application of the analyzed design pattern.

6.2 Experimental Setup

In this section, we describe the setup of our case study, including the feature-oriented SPLs that we analyze as well as the design patterns for which we developed detection techniques in Section 5.5.

Table 6.1: Overview of the analyzed feature-oriented SPLs

Program Name	#SLOC[1]	#FM[1]	Description
BerkeleyDB[3,4]	44 969	100	transactional storage engine
ChatSystem[3]	868	10	chat program
FeatureAMP[2]	2 497	29	mp3 music player
GameOfLife[3,4]	1 466	21	visual cellular automaton
GPL[3]	1 930	27	graph and algorithm library
Notepad[3]	1 751	13	text editor
Violet[3,4]	7 470	88	graphical UML editor

[1] SLOC: Source Lines of Code, FM: feature modules
[2] Developed in the context of the lecture *Software Product Lines: Concepts & Implementation* of Dr. Sandro Schulze, Institute of Software Engineering and Automotive Informatics, Technische Universität Braunschweig
[3] Obtained from the Fuji website: http://fosd.net/fuji
[4] Refactored from object-oriented legacy system

6.2.1 Feature-Oriented Software Product Lines

We conduct the case study on several feature-oriented SPLs that we list in Table 6.1. We selected a representative set of different sizes and domains from the available set of feature-oriented SPLs that have been developed in the context of research on FOP or teaching concepts of SPLs. Most of them, except *FeatureAMP*, which is a feature-oriented SPL developed in the context of a lecture, are exemplary product lines for FeatureHouse and Fuji and are established case studies for research on FOP.

Although there are more feature-oriented SPLs available, we can only use a small set for our study. Several SPLs, such as *TankWar*, contain syntactic or semantic errors, for example, missing field declarations so that some identifier references could not be resolved. This leads to parser errors when using FOPJaMoPP, which makes the creation of an 150% AST impossible. Other SPLs, such as *AHEAD* or *GUIDSL* do not contain a feature model, which is crucial for our analysis of feature collaborations. However, we argue, that the selected set of feature-oriented SPLs is representative to at least gain insights in the variability-aware application of design patterns.

With this set of product lines, we cover a variety of sizes, with the largest one consisting of 100 features and almost 45 000 lines of code, and the smallest consisting of 10 features and not even 900 lines of code. Moreover, we cover a variety of domains. *BerkeleyDB* is a large database backend system that has been refactored from the object-oriented legacy system. *ChatSystem*, *FeatureAMP* and *Notepad* are small and typical user applications for chatting, listening to music and editing text, respectively, all providing user interfaces and plenty of user interaction. *GameOfLife* is a small cellular automaton with a user interface. The *graph product line (GPL)* is a simple algorithm library and model for graph structures offering plenty of alternative and optional representations and extensions. *Violet* is a graphical editor for UML models such as state charts and class diagrams, offering fine-grained extension of the modeling palette.

6.2.2 Design Patterns

As we explained in Section 3.2, we do not want to concern all existing design patterns, but rather the patterns that have been identified to be suitable for product line design [2]. In particular, we analyze the feature-oriented application of the following design patterns:

Composite
The composite pattern (cf. Section 2.1.3) [16] is applied to dynamically compose part-whole hierarchies.

Observer
The observer pattern is used for decoupling of dependent objects (cf. Section 2.1.3) [16], which can dynamically register at a subject to be notified on the change of its state.

Strategy/Objectifier
The strategy pattern (cf. Section 2.1.3) [16] is used to objectify algorithms, which allows us to interchange similar, yet different algorithms at runtime. The objectifier shares the same structural conditions, however, its intent is more general than the strategy pattern since it is used to generally objectify behavior (cf. Section 2.1.3) [42].

Template Method
Using the template method pattern (cf. Section 2.1.3) [16], we can dynamically interchange steps of an algorithm, leading to runtime variability.

Most of these design patterns, the observer, strategy/objectifier and template method, are behavioral patterns. These patterns are concerned with varying behavior at runtime, mostly using inheritance in order to create variation. The composite pattern, as a structural pattern, does not use inheritance for variation, however, it is a general pattern to dynamically compose tree-structures of objects that can vary at runtime. We do not concern creational patterns at this point since they usually apply similar variation concepts as behavioral patterns in order to create different objects.

Other design patterns also focus on variability and modularity, however, these five patterns gain our main focus in this study. Besides being suitable for product line design, these patterns are also very common in object-oriented systems and, for the strategy and observer patterns we already have data on their existence in some of our regarded product lines [37]. In Section 5.5, we explain the realization of the detection for each of these patterns in detail.

6.3 Methodology

In order to analyze the usage of the selected set of design patterns in the selected set of feature-oriented SPLs, we follow the general workflow that we depicted in Figure 3.1 and explained in Section 3.2. In this section, we explain each step of our analysis in detail.

Parsing the SPLs

We start with FOPJaMoPP to parse the selected SPLs. To this end, we import all SPLs into an Eclipse instance, including all required class libraries so that every existing reference within the SPL can be resolved. We trigger the FOPJaMoPP parser that reads all Java files of the product line and then starts reference resolving and conversion to the 150% AST. We store the 150% AST of each SPL as a set of XMI files in order to reuse them for pattern detection.

Running the Pattern Detection

For each design pattern, we run Pattern Demon on each 150% AST of the selected SPLs. After the pattern detection, Pattern Demon prints the results, containing all involved elements (i.e., features, feature-oriented roles and methods) to the console and additionally stores them into a text file.

Eliminating False Positives and Duplicates

Although Pattern Demon already eliminates some false positives and duplicates automatically, we cannot reliably eliminate all of them because we cannot use behavioral analyses and check semantic conditions. Consequently, the set of matches still contains false positives and duplicates that have to be eliminated manually by reviewing each match and checking whether the detected matches fulfill the intent of the pattern. To eliminate duplicate matches, we try to find multiple design pattern matches containing similar collaborating elements. For instance, for the observer pattern, multiple, similar methods might be detected as the *addObserver* method. Imagine, for example, two matches for the observer that contain the same combination of `notify` and `update` methods, but one contains the `addObserver`, the other on the `removeObserver` playing the *addObserver* role (cf. Section 5.5). Both methods fulfill the static requirements of an *addObserver* method, which is why Pattern Demon detects them as separate matches. However, these are obviously duplicate results for the same pattern instance.

Moreover, we eliminate false positive matches by manually checking the semantics of the detected elements. In particular, for the regarded design patterns we need to check the following conditions:

Composite Is there actually a parent-child relation of related (i.e., inheriting) objects?
In particular, we need to check, whether the method playing the *addComponent* role actually is a method to add child objects to its enclosing object.

Observer Is there actually an event notification taking place?
For the observer pattern, we need to check whether the method playing the *notifyMethod* role actually calls the method playing the *update* role in order to notify the observer about a change of state.

Strategy/Objectifier Are the feature-oriented roles playing the *ConcreteStrategy* actually only objectified algorithms (i.e., strategy) or objectified behavior in general (i.e., objectifier)?
The intent of the strategy pattern is to encapsulate varying algorithms in separate objects [16]. The intent of the objectifier pattern is to encapsulate varying behavior of any kind in separate objects [42]. However, the detection technique detects almost every subclassing with two or more subclasses as a strategy/objectifier pattern, which is why we need to take a closer look at the concrete strategies and whether they only encapsulate specific algorithms/behavior.

Template Method Is there actually an algorithm skeleton where the hooks are specified by subclasses?
We need to check whether the method playing the *templateMethod* role of a detected pattern instance specifies an algorithm skeleton containing one or more calls to hook methods (i.e., abstract/empty methods or methods containing default behavior), which are then specified by subclasses.

While this step is tedious and time-consuming, it is necessary in order to get accurate results. After this step, our results only contain correct design pattern instances that match the structural as well as the semantic requirements, such that we can answer *Research Question 1* on how design patterns are decomposed along feature-oriented roles.

Constructing the Family Role Model

In order to answer *Research Question 2*, we take a closer look at the involved features for each instance of a regarded design pattern, and how these features are related in terms of the feature model. By combining the results for all detected instances, we capture the general collaborations of these features using an FRM (cf. Section 3.3), also taking the semantic requirements of the design pattern into account.

Deriving Guidelines and Application Rules

Using the FRMs of the different design patterns, we can answer *Research Question 3* by deriving guidelines and application rules for their variability-aware application as well as implications and consequences of such an application. For example, if multiple design pattern instances are implemented and decomposed the same way, we can reason why that is, what benefits such an application offers and what drawbacks it results in. With this step, we construct the variability-aware design pattern catalog.

6.4 Results

In this section, we present the results of our case study. We list the absolute numbers of detected design patterns in the selected feature-oriented SPLs in Table 6.2. For each design pattern, we list three categories:

- $\sum$ In this category, we list the mere results of the automated pattern detection using Pattern Demon. These results contain duplicate as well as false positives matches.
- $\sum_C$ In this category, we list the number of *correctly* identified design pattern instances after manually eliminating duplicate and false positive results.
- $\sum_D$ In this category, we list the number of *decomposed* design pattern instances that include at least two involved features contributing to the design pattern implementation.

We can see in Table 6.2 that we were able to detect instances of each design pattern in our case study. However, in some of the regarded SPLs, namely the two smallest SPLs *ChatSystem* and *Notepad*, we did not detect any of the regarded patterns. Moreover, we can see that we had to deal with plenty of false positives, especially for the observer pattern. Nevertheless, many of the detected design patterns are implemented across features. In the following, we give a detailed overview of the results for each design pattern, answering both *Research Question 1* and 2.

6.4.1 Composite

We only detected two correct results for the composite pattern in the two largest SPLs, *BerkeleyDB* and *Violet*. In the *BerkeleyDB*, the composite pattern is used to build up a general hierarchy of an event trace, where an `EventTrace` object consists of a comment and its successor in the chain of events of type `EventTrace`. In *Violet*, the composite pattern is used to express a hierarchy of nodes,

Table 6.2: Absolute numbers of detected design patterns

Program Name	Composite			Observer			Strategy			Template Method		
	$\sum$	$\sum_C$	$\sum_D$	$\sum$	$\sum_C$	$\sum_D$	$\sum$	$\sum_C$	$\sum_D$	$\sum$	$\sum_C$	$\sum_D$
BerkeleyDB	12	1	0	2516	–	–	55	–	–	94	11	2
ChatSystem	–	–	–	–	–	–	–	–	–	–	–	–
FeatureAMP	–	–	–	74	7	7	1	–	–	–	–	–
GameOfLife	–	–	–	2	1	1	3	1	1	–	–	–
GPL	–	–	–	12	–	–	–	–	–	1	–	–
Notepad	–	–	–	–	–	–	–	–	–	–	–	–
Violet	5	1	0	133	–	–	7	–	–	12	9	9

The results for the strategy pattern also contain the results for the structurally equivalent objectifier pattern
$\sum$ = Absolute number of design patterns automatically detected by Pattern Demon
$\sum_C$ = Absolute number of *correct* design patterns, after elimination of false positive and duplicate matches
$\sum_D$ = Absolute number of design patterns that are *decomposed* along features

where an `AbstractNode` object that implements the `Node` interface contains a list of child nodes of type `Node`. However, both of these composite pattern instances are implemented within one single mandatory feature, the *base* feature of *BerkeleyDB* and the *GraphUtility* feature of *Violet*. The *GraphUtility* feature is not actually declared mandatory in the feature model of *Violet*, but optional. However, in *Violet*, only the *base* feature that builds an empty Java frame is declared mandatory. Selecting any other feature leads to an inclusion of the *GraphUtility* feature, which is why we view it as mandatory. Since both pattern instances are not decomposed along features, we omit giving a concrete example at this point.

6.4.2 Observer

Even though we had to deal with plenty of false positive and duplicate results for the observer pattern, we managed to detect eight actual instances across two SPLs, all of which are of a decomposed nature. We identified the huge number of matches in the *BerkeleyDB* as well as in *Violet* as false positives and duplicates of these false positive results. In the following, we present the detected, decomposed pattern instances in *FeatureAMP* and *GameOfLife* separately for both SPLs.

FeatureAMP

In *FeatureAMP*, we managed to detect seven decomposed instances of the observer pattern, all of which are used to notify objects about events concerning the playback of a song. While they all implement the same `Listener` interface and are registered at the same *Subject*, which is the `AbstractAudioController`, we respect them as different instances of the observer pattern because each of them has their own `addXYZListener` / `removeXYZListener` and `notifyXYZListener` methods. The rest of the overall number of detections are duplicate matches of these seven instances, containing every possible combination of the different `addXYZListener` / `removeXYZListener` and `notifyXYZObserver` methods as they all only concern the unified `Listener` interface.

Because all seven observer instances concern similar events and share the same `Listener` interface as well as the same *Subject* object, they are implemented in a similar way. In Figure 6.1, we illustrate

Feature *BASE_FEATUREAMP*

```
1  public interface Listener<T> {  // Observer
2    public void update(T object);
3  }
4  public abstract class AbstractAudioController implements AudioController {
5    protected LinkedList<Listener<AudioController>> playListeners;  // SubjectField
6    public void addPlayListener(Listener<AudioController> l) {  // SubjectAdd
7      this.playListeners.add(l);
8    }
9    protected void notifyPlayListeners() {  // SubjectNotify
10     for (Listener<AudioController> l: this.playListeners) {
11       l.update(this);
12     }
13   }
14 }
```

Feature *MP3*

```
15 public class Mp3Controller extends AbstractAudioController {  // ConcreteSubject
16   public void play() {
17     /* ... */
18     this.notifyPlayListeners();
19   }
20 }
```

Feature *PLAYER_BAR*

```
21 public class PlayerBar {  // ConcreteObserver
22   class PlayListener implements Listener<AudioController> {
23     public void update(AudioController object) {
24       PlayerBar.this.playButton.setEnabled(false);
25       PlayerBar.this.pauseButton.setEnabled(true);
26       PlayerBar.this.stopButton.setEnabled(true);
27     }
28   }
29 }
```

Figure 6.1: Observer pattern instance `PlayListener` in *FeatureAMP*

the implementation of the `PlayListener` as an example for the observer pattern implementations in *FeatureAMP*. Using comments, we annotate, which parts of the detected pattern instance play which roles of the observer pattern in Figure 5.9.

In the feature *BASE_FEATUREAMP* in Line 1, the `Listener` interface is added playing the *Observer* role. The `AbstractAudioContoller` in Line 4 plays the *SubjectField* role by introducing a list of type `Listener` in Line 5. Moreover, an `addPlayListener` method is added in Line 6, which stores a parameter of type `Listener`, leading to the `AbstractAudioController` also playing the *SubjectAdd* role. Furthermore, the `notifyPlayListeners` method in Line 9 conforms to the *notifyMethod* role, which means that also the *SubjectNotify* role is played by the `AbstractAudioController`. Hence, all the *Subject*-related roles as well as the *Observer* role are played by feature-oriented roles of the feature *BASE_FEATUREAMP*.

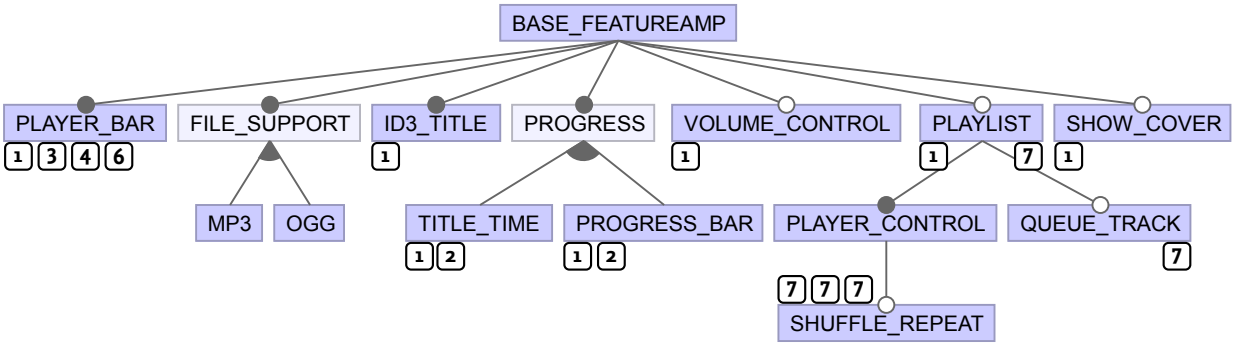


Figure 6.2: Extract of the feature model of *FeatureAMP* showing all features relevant to the observer patterns. Annotations show the *ConcreteObserver*s the introduces: 1. AudioListener, 2. TimeListener, 3. PlayListener, 4. PauseListener, 5. ResumeListener, 6. StopListener, 7. FinishedListener.

In the feature *MP3*, a feature-oriented role `Mp3Controller` inheriting from the `AbstractAudioController` is introduced in Line 15, which therefore plays the *ConcreteSubject* role for this pattern. With the `PlayListener` in Line 21 in feature *PLAYER_BAR*, a feature-oriented role implementing the `Listener` and therefore playing the *ConcreteObserver* is introduced as an internal class of the `PlayerBar`. Hence, the concrete subject as well as the concrete observer are both introduced in their own features. Other concrete subjects and concrete observers are introduced the same way.

Taking a closer look at all seven observer pattern instances in *FeatureAMP*, we observed that there are two feature-oriented roles playing the *ConcreteSubject* role for each of the pattern instances. In particular, each of the seven types of listeners can be registered at the `Mp3Controller` in feature *MP3* and the `OggController` in feature *Ogg*. Regarding the concrete observers, there are 17 different listeners playing the *ConcreteObserver* role, distributed across seven different features. However, all seven instances of the observer pattern share the same feature-oriented role `AbstractAudioController` of feature *BASE_FEATUREAMP* playing all *Subject*-related roles, as well as the unified `Listener` interface playing the *Observer* role for all pattern instances.

In particular, the following seven types of `Listener` objects playing the *ConcreteObserver* role exist, all having their own `add-`, `remove-` and `notify` methods in the `AbstractAudioController`. Hence, these seven listeners are the seven detected observer pattern instances:

1. `AudioListener`
2. `TimeListener`
3. `PlayListener`
4. `PauseListener`
5. `ResumeListener`
6. `StopListener`
7. `FinishedListener`

In Figure 6.2, we depict an extract of the feature model of *FeatureAMP* consisting of all features relevant to the seven observer pattern instances. We annotate, which features introduces a feature-oriented role that plays a *ConcreteObserver* role and to which instance of the mentioned observer

patterns the feature contributes by annotating the number of the listener. For the `ResumeListener`, the `PlayListener` of feature *PLAYER_BAR* is reused, which is why it is not annotated separately in the feature model.

The root feature *BASE_FEATUREAMP* introduces the *Subject* and *Observer* for all pattern instances, which is therefore always necessary. The *ConcreteSubject* role is played by feature-oriented roles introduced in the mandatory or-group *FILE_SUPPORT*, which means that at least one of them is always available.

Moreover, we see that there are plenty of different implementations of the `AudioListener` introduced in several features across the whole feature model, whereas other observers, such as the `Play-` or `StopListener`, are only introduced once in the feature *PLAYER_BAR*. However, there is no condition in which type of feature (e.g., mandatory, optional, or-group, etc.) a concrete observer is introduced.

GameOfLife

In *GameOfLife*, we detected one decomposed instance of the observer pattern used to notify the user interface on changes of the model. The other detection is a duplicate match of this pattern instance. In Figure 6.3, we depict an extract of the feature model of *GameOfLife* including the features involved in implementing the exemplary pattern instance that we depict in Figure 6.4.

The pattern is decomposed along two mandatory features, *ModelBase* and *GuiBase*. In the feature *ModelBase*, an interface `ModelObserver` is introduced, playing the *Observer* role of the pattern. Moreover, the feature-oriented role `ModelObservable` is introduced in Line 5, which plays all three *Subject*-related roles *SubjectField*, *SubjectAdd* and *SubjectNotify* by introducing a list of type `ModelObserver` in Line 6, the `attach` operation in Line 7 and the `notifyObservers` operation in Line 13. Moreover, the feature-oriented role `GODLModel` inheriting from `ModelObservable` is introduced in Line 22, thus, being the only object playing the *ConcreteSubject* role. In the feature *GuiBase*, the feature-oriented role `GolView` implementing the `ModelObserver` interface is introduced in Line 28. The `GolView` is the only object playing the *ConcreteObserver* role. Hence, the observer pattern in *GameOfLife* is implemented as a one-to-one relation of a single subject and its single observer implemented across two mandatory features.

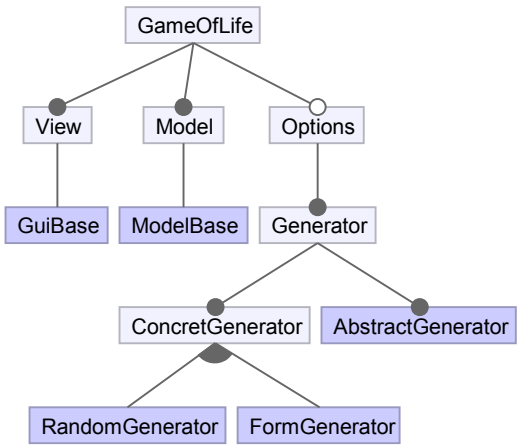


Figure 6.3: Feature model extract of *GameOfLife* showing features relevant to observer and strategy patterns

Feature *ModelBase*

```
1  public interface ModelObserver {  // Observer
2    public void update();
3  }
4  // SubjectField
5  public abstract class ModelObservable {
6    private List<ModelObserver> observers=new LinkedList<ModelObserver>();
7    public void attach(ModelObserver o){  // SubjectAdd
8      if (o == null) {
9        throw new IllegalArgumentException("Parameter is null");
10     }
11     observers.add(o);
12   }
13   public void notifyObservers(){  // SubjectNotify
14     Iterator it = observers.iterator();
15     while(it.hasNext()) {
16       ModelObserver x;
17       x = (ModelObserver) it.next();
18       x.update();
19     }
20   }
21 }
22 public class GODLModel extends ModelObservable {  // ConcreteSubject
23   public void setLifeform(int x, int y, int value) {
24     playground.set(x, y, value);
25     notifyObservers();
26   }
27 }
```

Feature *GuiBase*

```
28 public class GolView extends JFrame implements ModelObserver {  // ConcreteObserver
29   public void update(){
30     PPanel.display(model.getPlayground());
31     BTBar.update();
32   }
33 }
```

Figure 6.4: Observer pattern instance `ModelObserver` in *GameOfLife*

6.4.3 Strategy/Objectifier

We only have one single, correct match for the strategy pattern across all analyzed SPLs, and no results for the objectifier pattern. Even though Pattern Demon detected more than 60 overall matches, most of them are false positives not fulfilling the semantic requirements of the strategy and objectifier patterns, i.e., only encapsulating specific algorithms or behavior, respectively, instead of whole objects. In Figure 6.5, we illustrate the implementation of the detected strategy pattern in *GameOfLife*, the corresponding features are also contained in the feature model extract in Figure 6.3.

The `GeneratorStrategy` interface, introduced in Line 1 in feature *AbstractGenerator*, plays the *Strategy* role of the strategy pattern in Figure 5.11. A field of type `GeneratorStrategy` is held by the

Feature *AbstractGenerator*

```
1  public interface GeneratorStrategy { // Strategy
2    int getNext(int x, int y);
3  }
4
5  class GODLModel {
6    private GeneratorStrategy generator = null;  // ClientField
7    GODLModel(int xSize, int ySize, RuleSet rules) {
8      generators.add(new ClearGeneratorStrategy());
9    }
10   public void setGenerator(GeneratorStrategy generator) {
11     this.generator = generator;
12   }
13   public void generate() {  // ClientMethod
14     if (generator == null) {
15       generator = new ClearGeneratorStrategy();
16     }
17     /* ... */
18       generator.getNext(current.getX(), current.getY()));
19     /* ... */
20   }
21 }
22
23 // ConcreteStrategy
24 public class ClearGeneratorStrategy implements GeneratorStrategy {
25   public int getNext(int x, int y) {
26     return 0;
27   }
28 }
```

Feature *FormGenerator*

```
29 // ConcreteStrategy
30 public class FormGeneratorStrategy implements GeneratorStrategy {
31   public int getNext(int x, int y) {
32     /* ... */
33   }
34 }
35
36 class GODLModel {
37   GODLModel(int xSize, int ySize, RuleSet rules) {
38     FormGeneratorStrategy fgs = new FormGeneratorStrategy(/* ... */);
39     generators.add(fgs);
40   }
41 }
```

Figure 6.5: Strategy pattern instance GeneratorStrategy in *GameOfLife*

GODLModel, introduced in Line 5 in the same feature, which makes the GODLModel play the *ClientField* role. It also plays the *ClientMethod* role by calling the strategy method of the GeneratorStrategy in the method generate, introduced in Line 13. Moreover, in the feature *AbstractGenerator*, with the ClientGeneratorStrategy in Line 24, a feature-oriented role playing the *ConcreteStrategy* role is introduced. In the feature *FormGenerator*, another feature-oriented role, the FormGeneratorStrategy, playing the *ConcreteStrategy* role is introduced in Line 30. Moreover, there is a feature *RandomGenerator*, introducing the feature-oriented role RandomGeneratorStrategy, also playing the *ConcreteStrategy* role.

Hence, all *Client*-related roles of the strategy pattern in *GameOfLife* are introduced in one single feature-oriented role. In the same feature, the *Strategy* role as well as one default *ConcreteStrategy* is introduced. More feature-oriented roles playing the *ConcreteStrategy* role are introduced in other features. According to the feature model extract in Figure 6.2, in order to introduce this strategy pattern to a product, selecting the *AbstractGenerator* feature as well as at least one additional feature introducing a *ConcreteStrategy* is mandatory.

6.4.4 Template Method

Overall, we detected 20 correct instances of the template method pattern, of which eleven are decomposed along features. Two of these matches are located in the *BerkeleyDB*, the other nine matches are located in *Violet*. The matches we identified as false positives did not fulfill the semantic conditions of a template method pattern, i.e., providing a skeleton of an algorithm whose steps are partially defined in subclasses. In the following, we present the decomposed pattern instances detected in the case studies *BerkeleyDB* and *Violet*.

BerkeleyDB

We illustrate the implementation of a template method pattern in the *BerkeleyDB* in Figure 6.6. We depict the corresponding extract of the feature model in Figure 6.7. The pattern is implemented as follows. The feature-oriented role DaemonThread introduced in Line 2 in feature *base* plays both *Template* and *Hook* roles of the template method pattern in Figure 5.7. It introduces the *TemplateMethod* run in Line 3 that calls two *HookMethods* nDeadlockRetries and onWakeup, declared in the same feature-oriented role in Line 16 and Line 19, respectively. While onWakeup is an abstract operation that has to be defined by subclasses, nDeadlockRetries offers a default implementation.

Moreover, with the FileProcessor, a feature-oriented role playing the *HookChild* role and, thus, specifying both hook methods nDeadlockRetries and onWakeup, is introduced in the same feature. In the feature *INCompressor*, with the feature-oriented role INCompressor, another child class of the DaemonThread is introduced, specifying both hook methods, nDeadlockRetries and onWakeup, and, thus, playing the *HookChild* role. More of such feature-oriented roles playing the *HookChild* role are introduced in the features *CheckpointerDaemon* as well as in the derivative module *Derivative_Evictor_EvictorDaemon*, which depends on the inclusion of other features.

Hence, both *Template* and *Hook* are introduced in the same mandatory feature as well as one *HookChild*. Other feature-oriented roles playing the *HookChild* role are introduced in other optional features.

Feature *base*

```
1  // Template, Hook
2  public abstract class DaemonThread implements DaemonRunner, Runnable {
3    public void run(){  // TemplateMethod
4      while (true) {
5        /* ... */
6        int maxRetries=nDeadlockRetries();
7        do {
8          try {
9            /* ... */
10           onWakeup();
11           break;
12         } /* ... */
13       } /* ... */
14     }
15   }
16   protected int nDeadlockRetries() throws DatabaseException {  // HookMethod
17     return 0;
18   }
19   abstract protected void onWakeup() throws DatabaseException;  // HookMethod
20 }
21
22 class FileProcessor extends DaemonThread {  // HookChild
23   protected int nDeadlockRetries() throws DatabaseException {
24     return cleaner.nDeadlockRetries;
25   }
26   public void onWakeup() throws DatabaseException {
27     doClean(true,true,false);
28   }
29 }
```

Feature *INCompressor*

```
32 public class INCompressor extends DaemonThread {  // HookChild
33   protected int nDeadlockRetries() throws DatabaseException {
34     return env.getConfigManager().getInt(EnvironmentParams.COMPRESSOR_RETRY);
35   }
36   public synchronized void onWakeup() throws DatabaseException {
37     if (env.isClosed()) {
38       return;
39     }
40     this.hook403();
41     doCompress();
42   }
43 }
```

Figure 6.6: Template method pattern instance DaemonThread in *BerkeleyDB*

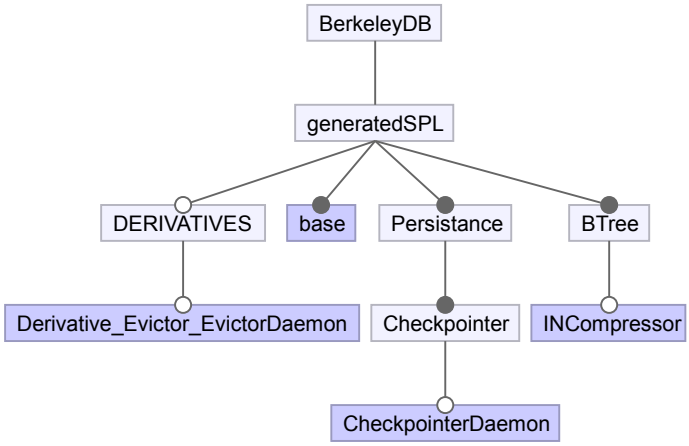


Figure 6.7: Extract of the feature model of *BerkeleyDB* showing all features relevant to the template method pattern

Violet

In *Violet*, we detected a number of decomposed template method patterns, however, some of them are related having different template methods calling the same hook methods. In Figure 6.8, we illustrate the implementation of an instance of the template method pattern in *Violet*. In Figure 6.9, we depict the corresponding feature model extract including the relevant features for this template method pattern instance in *Violet*.

In this template method pattern instance, the pattern is used for drawing nodes of different shapes. The *AbstractNode* feature introduces the feature-oriented role `AbstractNode` that plays both *Template* and *Hook* by specifying the algorithm skeleton in the `draw` method in Line 2, containing a call to the *HookMethod* `getShape` that is declared in Line 6. A total number of eight feature-oriented roles introduced in seven different features inherit from the `AbstractNode`, implementing the `getShape` method and, thus, playing the *HookChild* role, such as the `PackageNode` introduced in Line 11 in feature *ClassDiagram*, or the `FieldNode` introduced in Line 19 in feature *ObjectDiagram*.

Peculiar about the implementation of template method patterns in *Violet* is that some template methods share common hook methods. For example, the feature-oriented role `SegmentedLine-Edge` in the feature *DiagramSupport* introduces five different operations playing the *TemplateMethod* role, such as `draw` or `getBounds`, all calling the same hook method `getPoints` that is introduced in the same feature-oriented role. *HookChilds* are the feature-oriented roles `CallEdge` and `ReturnEdge` in feature *SequenceDiagram* as well as `ClassRelationshipEdge` in feature *DiagramSupport*.

According to the feature model in Figure 6.9 that includes all features relevant to the implementation of this template method pattern instance, the feature-oriented roles playing the *HookChild* role are all implemented in independent, optional features across the feature model, contributing to different model types, such as the class or object diagram. The roles *Template* and *Hook* are played by the same feature-oriented role. This observation holds for all detected instances of the template method pattern in *Violet*.

Feature *AbstractNode*

```
1  public abstract class AbstractNode implements Node {  // Template, Hook
2    public void draw(Graphics2D g2) {  // TemplateMethod
3      Shape shape = getShape();
4      /* code using shape */
5    }
6    public Shape getShape() {  // HookMethod
7      return null;
8    }
9  }
```

Feature *ClassDiagram*

```
11 public class PackageNode extends RectangularNode {  // HookChild
12   public Shape getShape() {
13     GeneralPath path = new GeneralPath();
14     path.append(top, false);
15     path.append(bot, false);
16     return path;
17   }
18 }
```

Feature *ObjectDiagram*

```
19 public class FieldNode extends RectangularNode {  // HookChild
20   public Shape getShape() {
21     if (boxedValue)
22       return valueBounds;
23     else
24       return null;
25   }
26 }
```

Figure 6.8: Template method pattern instance `AbstractNode` in *Violet*

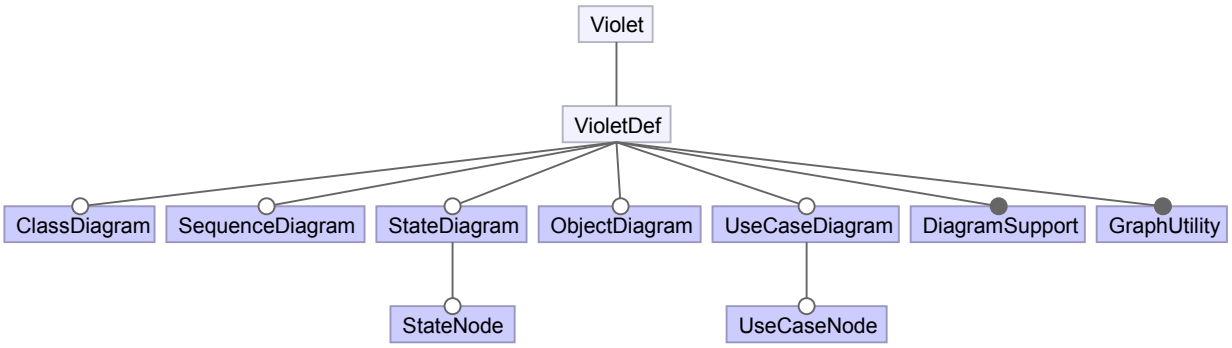


Figure 6.9: Extract of the feature model of *Violet* showing all features relevant to the template method pattern instance `AbstractNode`

6.5 Discussion

In this section, we discuss our results of the last section. In particular, we aim at answering *Research Question 3* by taking a closer look at the results and discussing their implications. Moreover, we briefly evaluate the automated design pattern detection using Pattern Demon.

6.5.1 Implications of the Results

The results of the case study are promising, such that, in the following, we can discuss implications and consequences, which we do separately for each design pattern.

Composite

For the composite pattern, we did not detect any decomposed instances. This might result from the fact that the composite pattern does not achieve variability in the way in which the observer, strategy or template method patterns do. In the composite pattern, we can create a hierarchical structure of atomic and composite components. This structure can be changed dynamically, however, there are no static variations between two different hierarchical structures, which means, we do not introduce separate classes in order to create different hierarchies.

To create part-whole hierarchies, the composite pattern in its common form mainly introduces two classes, a `Component` as well as a `Composite`, inheriting from the `Component` and holding references to an arbitrary number of `Component` objects. In the two correct composite pattern matches of our case study, both, the `Component` as well as the `Composite`, are introduced in the same containing feature. Because there is no decomposed match, we can only reason about whether a decomposition of the composite pattern would even be beneficial.

On the one hand, it might be possible to add an atomic `Component` object without the intent of creating a hierarchical structure. For a subsequent feature, the requirement of allowing composite structures and, thus, creating a part-whole hierarchy might emerge, such that this feature later adds the `Composite` functionality. This way, a decomposition of the composite pattern would be achieved by refining a `Component` with a composite structure.

On the other hand, looking at the decomposition of the other design patterns as presented in the results, the existence of such a decomposition of the composite pattern is not very likely. For other design patterns, only concrete subclasses of abstractions are encapsulated by their own features, whereas the abstraction and, therefore, the basic implementation of the design pattern is always introduced within one single feature. For the composite pattern, both `Component` and `Composite` classes are necessary in order to actually implement the pattern. Dedicated concrete subclasses varying the behavior are usually not part of a composite pattern. This would imply that the implementation of a composite pattern should always be performed within one feature.

Nevertheless, even though it is not a usual implementation, but rather a variation of the composite pattern, introducing different kinds of leaf and composite objects is possible, which would imply a possible decomposition for different variations of leafs and composites. However, we did not detect any instance of such a variation, which is why we can only reason on its application in the context of FOP. Hence, based on our results, we argue that the composite pattern does not appear to be suitable for decomposition in the context of FOP.

Observer

All detected instances of the observer pattern are implemented in a decomposed fashion. While in *GameOfLife*, there is only a one-to-one relation of a single subject (model) with a single observer (view), there are seven observer pattern instances in *FeatureAMP*, distributed across the whole feature model.

In both cases, *GameOfLife* as well as *FeatureAMP*, the abstractions (i.e., the *Subject* and the *Observer* roles) are introduced within one single base feature. This is reasonable because the different subject roles (i.e., *SubjectField*, *SubjectAdd* and *SubjectNotify*) can be seen as one unit of functionality. Introducing a list of observer objects would not be sensible, if it is never used. And we would not need an observer interface without a subject to observe. Hence, in our findings, there is always a feature that encapsulates the whole functionality required to realize event notification for the respective application scenario (e.g. updating the view on model changes). There is never any decomposition of the abstract functionality of an observer pattern.

The decomposition takes place for the concrete subclasses of both subject and observer. While the *ConcreteSubject* in *GameOfLife* is also introduced in one feature together with the *Subject* and *Observer* roles, in *FeatureAMP*, two *ConcreteSubject* roles are introduced in two different features of a mandatory or-group. Hence, if the observer pattern is introduced, the existence of at least one *ConcreteSubject* role is necessary.

However, feature-oriented roles playing a *ConcreteObserver* are not necessarily required. They have to exist, in order for the observer pattern to actually be used, however, they do not have to be part of every configuration in which the observer pattern abstractions are present. Usually, a *ConcreteObserver* does not add to the functionality of the *Subject*, but rather to another functionality that depends on the state of a *Subject*. Hence, there might be configurations, in which no dependent objects exist. Because of this, in our findings, a *ConcreteObserver* is introduced in another feature as the general event notification functionality. Moreover, most feature-oriented roles playing the *ConcreteObserver* role are enclosed by optional features, especially in *FeatureAMP* (cf. Figure 6.2). Because it depends on the state of a *Subject*, the general event notification has to be existent for the *ConcreteObserver* to be introduced.

Decomposing an observer pattern in this manner leads to a caller-callee interaction of the feature introducing the *notifyMethod* and the feature introducing the *updateMethod*. Hence, decomposing an observer pattern appears to be highly beneficial. We can achieve a clean separation of concerns while preserving dependencies of objects. Moreover, we reduce the coupling of features. Only features introducing feature-oriented roles playing the *ConcreteObserver* role depend on the feature introducing the event notification (i.e., *Subject* and *Observer*) as well as on features introducing the respective feature-oriented roles playing the *ConcreteSubject* role.

Strategy / Objectifier

Since the strategy and objectifier patterns have a similar intent and an equal structure, we address them together. We only detected one instance of the strategy and none of the objectifier pattern, however, because of their similarity, we argue that the implications hold for both patterns.

For the strategy pattern, similar implications as for the observer pattern hold. A *Client* without a *Strategy* cannot be a *Client* in the sense of a strategy pattern. Since the *Strategy* is encapsulated behavior of a *Client*, it cannot exist without a *Client* present. The findings in *GameOfLife* reflect these

implications in the way that both abstraction, *Client* and *Strategy*, are introduced within the same feature.

While a feature-oriented role cannot play the *Client* role without a feature-oriented role playing the *Strategy* role, the class, to which the *Client* role is introduced, can exist before the strategy pattern is introduced. This way, a class can be refined with a *Client* role of a strategy pattern. In *GameOfLife*, this is actually the case. The class `GODLModel` is introduced in the feature *ModelBase*, which is the first feature to be applied according to the feature order. The *AbstractGenerator*, introducing the abstractions for the strategy pattern, is the fourth feature in the feature order, thus, refining the `GODLModel` with the *Client* role and introducing a feature-oriented role playing the *Strategy* role. Hence, an existing class is extended by functionality that is factored out using the strategy pattern.

Regarding the *ConcreteStrategy* role, in contrast to a *ConcreteObserver*, there must always be at least one `ConcreteStrategy` if the strategy pattern is introduced because the `ConcreteStrategy` encapsulates the necessary behavior of the feature-oriented role playing the *Client* role. Moreover, feature-oriented roles playing the *ConcreteStrategy* are not independent units as a *ConcreteObserver*, but rely on the *Client* because they factor out varying client behavior. Hence, features introducing a *ConcreteStrategy* are not distributed across the whole feature model, but rather encapsulated as a subgroup or with the same parent feature as the feature introducing the *Client*.

The alternative of applying the strategy pattern would be to refine the varying client functionality with the desired behavior, which could cause plenty of unwanted structural feature interactions. Using this decomposition, only a caller-callee interaction of the client and the strategy takes place. Hence, decomposing a strategy pattern allows to encapsulate varying client behavior in separate features while reducing the risk of feature interactions.

Template Method

For the template method pattern, we detected 20 matches, of which eleven are of a decomposed nature, leading to the assumption that the template method pattern must be beneficial to be implemented in the context of FOP.

In *BerkeleyDB*, the template method pattern is applied to build general skeletons for a `DaemonThread` as well as for a `LockManager`, which are both domain-independent concepts. In *Violet*, the pattern is applied to build skeletons for drawing and handling nodes and edges of different types and shapes and, thus, for a domain-specific implementation. In both cases, the template method pattern is used to provide general behavior in mandatory features.

To this end, the *Template* as well as all corresponding *Hook* roles are always introduced together within the same feature. Consequently, there might not be a need for refining the operation playing the *templateMethod* role to call new methods playing the *hookMethod* role. The general implementation of the *templateMethod* always appears to be known when first introduced.

In contrast, concrete implementations of the operations playing the *hookMethod* roles are distributed across different features and functionalities. In *BerkeleyDB*, different functionalities of different features employ a `DaemonThread` or a `LockManager`. In *Violet*, different types of diagrams introduce their own nodes and edges.

Generally, a concrete implementation playing the *HookChild* role is necessary in order to use the *templateMethod*, however, default implementations can be provided, either using concrete methods playing the *hookMethod*, or making the existence of a default *HookChild* implementation mandatory.

The first is done in *BerkeleyDB* with the *hookMethod* `nDeadlockRetries`. The latter is applied in *Violet*, where the mandatory feature *DiagramSupport* introduces a `RectangularNode`, which is used as basis for other node types. Because the *HookChild* roles require the *Template* and *Hook* roles, and these *Template* and *Hook* roles realize general concepts, both *Template* and *Hook* are introduced in mandatory features.

The template method pattern appears to be highly suitable for decomposition, especially when implementing general concepts that are relevant for a number of features as skeletons in high-level features. The varying parts of these skeletons can be implemented in low-level features. Hence, in these low-level features, we can concentrate on the essential functionalities necessary to implement the feature, such as only defining the shape of the respective node instead of providing the complete operation to draw the node (cf. Figure 6.8).

Moreover, similar to the strategy pattern, we reduce the risk of unwanted structural feature interactions by only relying on a caller-callee relation of the involved features. Consequently, we benefit from applying and decomposing the template method pattern along features in the context of FOP.

6.5.2 Evaluating Pattern Demon

Using Pattern Demon, we were able to detect a number of decomposed design patterns. However, we had to manually perform a semantics check on each match because Pattern Demon detected plenty of false positive (i.e., matches that are not actual pattern instances) and duplicate matches. This high number of false positives, especially for the observer pattern, is caused by only using static analysis for design pattern detection. Using static analysis, we can only check structural as well as data and control flow constraints. Moreover, we had to describe the constraints for each design pattern as general as possible because design patterns usually do not resemble a definite structure, but are rather general, informal descriptions on how to solve a specific problem.

If we described the design pattern too specifically, we would increase the number of true negatives (i.e., actual pattern instances that are not detected). However, describing the design pattern too generally increases the number of false positives. Since detecting as many pattern instances as possible was important to derive guidelines, we settled for a high number of false positives that we eliminated manually.

Regarding duplicate results, we already combine specific duplicates automatically, however, we cannot reliably eliminate all of them. For instance, the subject of an observer pattern usually contains two methods, an `addObserver` and a `removeObserver`. Not being too restrictive, we only checked for the `addObserver` operation to be present. However, a `removeObserver` method also fulfills all static conditions of the *addObserver*. The `Observer` is passed as an argument, and it is usually used at the right hand side of an argument or passed to another method (cf. Section 5.5.3). Consequently, a new pattern instance containing the `removeObserver` method playing the role of the *addObserver* is detected. Moreover, when implementing different observer pattern instances that all rely on the same `Listener` interface, as done in *FeatureAMP*, the different `add`, `remove` and `notify` operations cannot be distinguished. This leads to each combination of these operation being detected as a separate match.

Furthermore, the single detected match of the template method pattern in the *GPL* is peculiar because an `original`-call is identified as a call to a hook method. This is caused by FOPJaMoPP parsing and resolving an `original`-call as a method call to all declarations of the enclosing method.

In addition, we had problems conducting the pattern detection on large models, especially for large design patterns like the observer pattern. Because the observer pattern description has many parameters (cf. Section 5.5.3), plenty of combinations of these parameters have to be stored by EMF-IncQuery. Since the memory consumption of EMF-IncQuery does not seem well-engineered, a common computer with 4GB of RAM was not sufficient, using a server with 48GB of RAM was necessary to let the observer pattern detection run smoothly on large models, such as the *BerkeleyDB*.

Nevertheless, the overall picture of the results of Pattern Demon looks promising. We detected several, genuine design pattern implementations, most of which are decomposed along features.

6.6 Threats to Validity

While we carefully conducted our case study, it still suffers threats concerning the generalization of the results. In the following, we address the different threats to validity.

6.6.1 Construct Validity

While we describe design patterns for the detection process as generally as possible using role modeling and, thus, concentrate on the essential characteristics of a design pattern, there is no standard description of these characteristics [13]. Each design pattern is described informally, which means, there is no formality about the characteristics of a pattern that could be followed during the detection. Moreover, each pattern can be implemented in a variety of ways, which makes formalizing the unique characteristics of a pattern even harder.

Consequently, we answer our research questions according to our understanding of the analyzed design patterns, which might be different to the understanding of others. Because of this, the answers to the research questions are largely dependent on the understanding and description of what constitutes a specific pattern and what does not.

6.6.2 Internal Validity

We formalized each design pattern fairly specifically based on the static detection approach of Heuzeroth et al. [20]. Using this fairly specific description for our detection, we most likely reject actual instances of design patterns because they varied too much according to our characteristics (i.e., true negatives). Hence, our descriptions of the design patterns might not completely represent the general understanding of the patterns, but rather is reliant on our static detection technique. Other descriptions of design patterns might lead to other instances of design patterns being detected, which, consequently, might lead to different results.

Moreover, we were not able to prove the correctness of FOPJaMoPP, other than testing it on minimal examples and reviewing samples of the analyzed feature-oriented SPLs. The same holds for Pattern Demon, whose accuracy and correctness could not be tested due to a lack of documentation on the use of design patterns. Because both approaches are specifically tailored to work with feature-oriented SPLs, using well-documented object-oriented case studies was not an option. Nevertheless, we carefully reviewed the code of both tools and tested both with a variety of toy examples. Moreover, we actually detected several actual design pattern instances, so that we argue that both tools are at least trustworthy enough to gain reliable insights in the application of variability-aware design patterns. Still, errors in the tools might distort the results.

6.6.3 External Validity

Due to a lack of feature-oriented SPLs that fulfill our requirements (cf. Section 6.3), we could only analyze a small set of SPLs, which might forfeit the generalizability of our results. However, we argue that we cover a variety of sizes and relevant domains with the selected set of SPLs, increasing the probability of detecting design patterns and also increasing the generalizability of the results.

Also, some of the considered SPLs are refactored from object-oriented legacy systems while others are developed from scratch using FOP. Because of this, the comparability between SPLs of these two development approaches might be impaired, however, we detected design pattern instances in both, refactored SPLs and SPLs developed from scratch.

Moreover, we only considered five design patterns although a huge set of patterns exists. We argue that with this set of design patterns, we cover common and important variability patterns, which are relevant for this work. However, there are plenty of other design patterns that concern modularity and variability and that have to be analyzed in order to generalize our results.

6.6.4 Reliability

As mentioned above, describing design patterns in a different, more specific or more generalized way, as well as using other approaches to detect design patterns, might lead to different results, even when using the same case studies [13]. However, we carefully developed our design pattern detection to use specific descriptions of general characteristics of design patterns, eliminating false positives by a manual semantics check. Hence, our findings should be reproducible when using the same case studies.

On the other hand, analyzing other SPLs or the use of other design patterns might lead to completely different results because it might reveal design pattern applications that we did not detect. Hence, our results might be largely dependent on the selection of SPLs and design patterns for the case study. However, since we detected several actual, genuine design pattern instances that are not matter of interpretation, we argue that we developed a reliable approach. To verify the findings, more SPLs and design patterns should be analyzed.

6.7 Summary

In this chapter, we conducted a case study on several feature-oriented SPLs in order to reveal the variability-aware application of the five design patterns *composite*, *objectifier*, *observer*, *strategy* and *template method*. We detected instances of all patterns except the objectifier pattern. Except for the composite pattern, most of the detected matches were decomposed along features, allowing us to analyze the feature collaborations applied to implement the pattern.

We reasoned that the *composite* pattern might not be suitable for decomposition because of its nature of not providing different concrete classes for varying hierarchies, but rather dynamically employing the same classes. In contrast, the *observer*, *strategy* and *template method* patterns appear to benefit from decomposition. Moreover, the implementation of the feature-oriented SPL benefits from applying these patterns in a decomposed manner. Using the *observer* pattern, a clear separation of concerns and loose feature coupling can be achieved. Applying the *strategy* pattern allows the encapsulating of varying client behavior in different features. The *template method* pattern makes abstraction across features possible by providing a general skeleton in a high-level feature that is

specified in low-level features. This allows for specific features to only concern their essential functionality.

Moreover, we discovered that dependent parts of a pattern are usually introduced as a whole. *Subject* and *observer*, *strategy* and *client* as well as *template* and *hooks* are always introduced within one feature. Only concrete subclasses, such as the *concrete subjects* and *observers*, *strategies* and *hook method* implementations, are usually encapsulated by different features.

Furthermore, with applying decomposed design patterns, the risk of unwanted structural feature interaction is reduced compared to not applying design patterns, by only relying on caller-callee relations instead of refinements.

Using these results and implications, we can derive guidelines and application rules for the variability-aware application of design patterns, leading to a catalog of variability-aware design patterns in the next chapter.

7 Towards a Variability-Aware Design Pattern Catalog

In the last chapter, we conducted a case study on the variability-aware application of design patterns in the context of FOP. In this chapter, we use the results and discussion of the case study. For each analyzed design pattern, we derive family role models, which we introduced in Section 3.3, and create a catalog page, which we introduced in Section 3.4, including guidelines and application rules for the variability-aware design pattern application. In Figure 7.1, we depict a legend for the possible feature collaborations in the FRM.

(a) Prohibition – A and B not played by same feature

(b) Equivalence – A and B played by same feature

(c) Requires – A requires B

Figure 7.1: Legend for feature collaborations in family role models

7.1 Deriving Family Role Models

In the following, we derive the feature role models for each analyzed design pattern. We show the respective FRMs in the corresponding catalog pages in Section 7.2.

7.1.1 Composite

We depict the FRM for the composite pattern in the corresponding catalog page in Figure 7.2. For the composite pattern, we did not detect any decomposed matches. However, we detected two matches, one in the *BerkeleyDB* and one in *Violet*, where all roles are introduced within a single feature. We reasoned that the composite pattern might not be suitable for decomposition. From the results of the other design patterns, we have learned that decomposition usually takes place for varying concrete subclasses of an abstraction, for instance, for concrete observers. As the composite pattern usually does not apply inheritance in order to encapsulate variation, but rather object composition, such a decomposition is not possible. While we argue that such a decomposition could, nonetheless, exist, we can only reason on its decomposed application. Hence, as we show in Figure 7.2, for now, we suggest implementing all roles within one feature, as it is done in both detected matches.

7.1.2 Observer

We depict the FRM for the observer pattern in the corresponding catalog page in Figure 7.3. We base the feature role model on the several decomposed matches that we detected in the case study, which are all decomposed in a similar manner. The abstractions, *Subject* and *Observer*, introducing the general event notification mechanism, are always encapsulated within one single feature, which

we argue is reasonable. Because this is a coherent unit of functionality, we suggest to implement the *Observer* as well as all *Subject*-related roles with one feature. If there are several feature-oriented roles playing the *ConcreteSubject* role, as in *FeatureAMP*, they might be implemented in several features since they might provide distinguishable functionalities. In contrast, if there is only one *ConcreteSubject*, it could be introduced within the feature introducing *Subject* and *Observer*, as in *GameOfLife*.

Usually, the subject of an observer pattern is independent of any feature-oriented role playing the *ConcreteObserver* role, since *ConcreteObservers* usually do not add subject functionality, but rather new functionality that depends on the subject. Hence, we suggest introducing such feature-oriented roles playing the *ConcreteObserver* in different features distributed across the feature model. However, these features introducing a *ConcreteObserver* require the abstractions, the *Subject* and *Observer*, as well as at least one *ConcreteSubject*.

7.1.3 Strategy / Objectifier

We illustrate the FRM for the strategy/objectifier patterns in the corresponding catalog page in Figure 7.4. As discussed, a client without a strategy cannot be a client in the sense of a strategy pattern, however, the class can exist beforehand and be refined with the *Client* role. Because the *Strategy* encapsulates *Client* behavior, it cannot exist for itself. Its existence relies on the *Client* role. Hence, it must be introduced together with the *Client* role.

We detected that feature-oriented roles playing the *ConcreteStrategy* role are introduced in single features because they contribute to the *Client* functionality with distinguishable sub-functionality. Also, we detected that a default *ConcreteStrategy* can be introduced together with the *Client* and *Strategy* roles. Hence, we suggest separate features for feature-oriented roles playing the *ConcreteStrategy* role except for a default *ConcreteStrategy*.

Because the *ConcreteStrategy* totally relies on the *Client* functionality that is extended, in addition to the FRM, we suggest constraints on the feature model. A *ConcreteStrategy* should be introduced in a subfeature of the *Client* or with the same parent feature as the feature introducing the *Client*.

7.1.4 Template Method

We depict the FRM for the template method pattern in the corresponding catalog page in Figure 7.5. We detected several similar matches, which we used to derive the FRM. In all matches, the template method pattern is applied to abstract general, domain-independent or domain-specific, concepts, such as realizing a daemon thread or the general drawing functionality of nodes in a graphical editor.

Even though FOP allows for the `templateMethod` and a `hookMethod` to be provided in different features, this is never the case in the case study (and introduces feature dependencies). The *Template* role and all corresponding *Hook* roles are always introduced in the same feature. Hence, the skeleton of the algorithm always seems to be known when the `templateMethod` is introduced. Consequently, we suggest implementing *Template* and the corresponding *Hooks* within one feature.

In all detected matches, the specific implementations of the *hookMethods* are distributed across the feature model and mostly contained in optional features. These *HookChild* roles contribute to different functionality, reusing the abstractions introduced with the *Template* role.

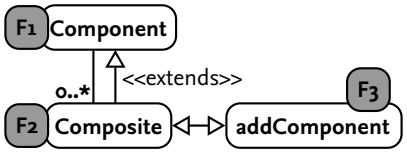
7.2 Catalog Pages

Composite pattern

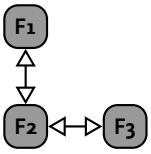
Intent

Introduce a part-whole hierarchy encapsulating coherent functionality.

Solution



(a) Role model for the composite pattern annotated with features



(b) Role model for participating features of the composite pattern

We depict the structure of the solution above. In Figure 7.2a, we show the different roles with their introducing features, whereas in Figure 7.2b, we illustrate the relationships and dependencies of these features. We make the following suggestion:

- Introduce both, *Component* and *Composite* roles of Figure 7.2a, together in one feature. Hence, in Figure 7.2b, *F1* and *F2*, introducing *Component* and *Composite*, respectively, are equivalent. *F2* and *F3* must be equivalent because the *Composite* (*F2*) introduces the *addComponent* (*F3*).

Consequences

Using this solution, the following consequences are implied:

+ Clear encapsulation of coherent, tree-structured data and functionality.

– No decomposition and, hence, no feature modularity.

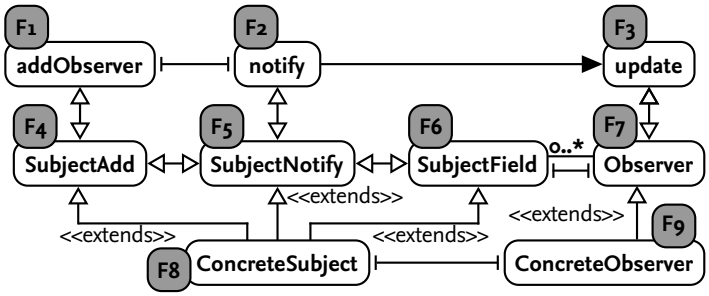
Figure 7.2: Catalog page for the composite pattern

Observer pattern

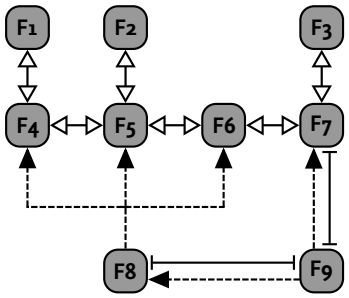
Intent

Create an event-notification of observing objects while modularizing concrete subjects and observers by functionality.

Solution



(a) Role model for the observer pattern annotated with features



(b) Role model for participating features of the observer pattern

We depict the structure of the solution above. In Figure 7.3a, we show the different roles with their introducing features, whereas in Figure 7.3b, we illustrate the relationships and dependencies of these features. We make the following suggestions:

- Introduce all abstracting *Subject*-related as well as *Observer*-related roles of Figure 7.3a together in one feature. The *Subject* is introduced using three roles, the *SubjectAdd*, *SubjectNotify* and *SubjectField* roles (feature roles *F4* – *F6* in Figure 7.3b). Hence, all three feature roles *F4*, *F5* and *F6* are equivalent. This also implies an equivalence to *F1* and *F2*, introducing the *addObserver* and *notify* roels. Moreover, the *Observer* role with its *update* role should be introduced in the same features, implying an equivalence of *F3* and *F7* as well.
- Modularize different *ConcreteSubjects* (*F8*) by functionality, such that at least one is mandatory if the *Subject* (*F4* – *F6*) exists.
- Modularize different *ConcreteObservers* (*F9*) by functionality. While requiring for a *ConcreteSubject* (*F8*) and the *Observer* (*F7*) to exist, no other relations to other features are necessary.

Consequences

Using this solution, the following consequences are implied:

+ The observer's functionality is encapsulated independently of subjects.
+ Modularization of abstractions and concrete implementations is allowed, easing the extensibility for new *ConcreteSubjects* and *ConcreteObservers*.
– The *feature-optionality problem* (cf. Section 2.2.3) is introduced if both, *ConcreteSubjects* and *ConcreteObservers*, are optional. Adding a *ConcreteSubject* might lead to unexpected updates for *ConcreteObserers*, which would have to be refined in order to understand new updates.

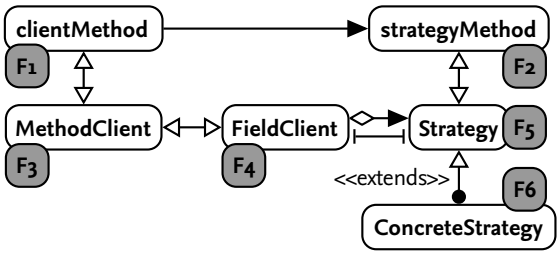
Figure 7.3: Catalog page for the observer pattern

Strategy pattern

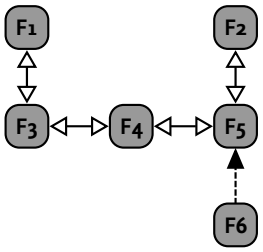
Intent

Encapsulate interchangeable algorithms/behavior while allowing compile-time selection.

Solution



(a) Role model for the strategy pattern annotated with features



(b) Role model for participating features of the strategy pattern

We depict the structure of the decomposed pattern above. In Figure 7.4a, we show the different roles with their introducing features, whereas in Figure 7.4b, we illustrate the relationships and dependencies of these features. We make the following suggestions:

- Introduce *Client* and *Strategy* of Figure 7.4a together in one feature. Hence, *F3* and *F4* as well as *F5* in Figure 7.4b are equivalent, which also implies the equivalence of *F1* and *F2*.
- Introduce *ConcreteStrategies* separately while requiring the *Strategy*.
- Make at least one *ConcreteStrategy* mandatory if *Client* and *Strategy* are introduced.
- Introduce *Client* and *Strategy* as well as *ConcreteStrategies* in the same feature group (i.e., common parent feature).

Consequences

Using this solution, the following consequences are implied:

+ Configure a *Client* at compile-time with varying behavior using one or more *ConcreteStrategies*.
+ Ease extensibility for new *ConcreteStrategies*.
– *Client* must be made aware of existing *ConcreteStrategies*. To this end, the feature template method (cf. Section 2.5) might be suitable, filling an operation to register strategies with each feature introducing a *ConcreteStrategy*.

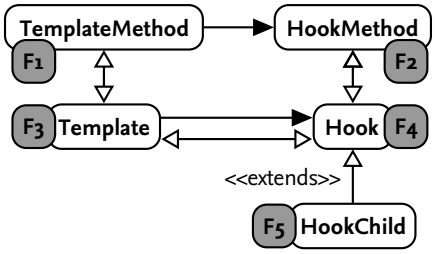
Figure 7.4: Catalog page for the strategy pattern

Template method pattern

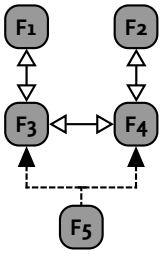
Intent

Encapsulating invariant parts of an algorithm in one a feature, factoring out varying parts to subsequent features.

Solution



(a) Role model for the template method pattern annotated with features



(b) Role model for participating features of the template method pattern

We depict the structure of the solution above. In Figure 7.5a, we show the different roles with their introducing features, whereas in Figure 7.5b, we illustrate the relationships and dependencies of these features. We make the following suggestions:

- Introduce *Template* and all *Hooks* of Figure 7.5a together in one feature, which implies an equivalence of $F3$ and $F4$ in Figure 7.5b.
- Introduce *HookChilds* almost independently across feature model, only *Template* ($F3$) and at least one *Hook* ($F4$) have to exist.

Consequences

Using this solution, the following consequences are implied:

+ Introducing general behavior in high-level features, letting low-level features implement domain-specific behavior, thus, concentrating on the essentials in low-level features.
+ Implementation similar to frameworks, exploiting the *Hollywood Principle* (cf. Section 2.1) across features by letting subsequent features fill domain-specific parts of an algorithm.
– Implicit implementation dependency: A feature introducing a *HookChild* must provide implementations for the *Hook* methods.

Figure 7.5: Catalog page for the template method pattern

7.3 Summary

In this chapter, we created catalog pages (cf. Section 3.4) for the variability-aware application of the analyzed design patterns. We used the results of the case study to derive a family role model (cf. Section 3.3) for each pattern and derive guidelines and application rules for the variability-aware application.

We described the usual decompositions of the analyzed design patterns. For the composite pattern, we did not detect any decomposed instance, hence, we suggest introducing the pattern within one single feature. For all other analyzed patterns, *objectifier*, *observer*, *strategy* and *template method*, we detected decomposed instances of a similar nature. In general, encapsulating the abstractions, which introduce the general concepts of the pattern, within one feature appears to be sensible. Concrete implementations, on the other hand, are generally decomposed along features. Such a decomposition of these design patterns appears to be sensible due to an increase in modularity and reuse by decoupling specific implementations from abstraction. Applying such a decomposition to these patterns allows fine-grained customization by exploiting runtime variability offered by design patterns. However, when decomposing design patterns, feature interactions and especially the feature-optionality problem (cf. Section 2.2.3) need to be taken into account.

8 Discussion

In the course of this work, different aspects and issues arose regarding the general approach as well as the results. In this chapter, we discuss and reason about these aspects. First, we point out and discuss limitations regarding the expressiveness of family role models in Section 8.1. Next, we compare the results of this work to the results of our prior case study on design patterns in feature-oriented SPLs [37] in Section 8.2. Finally, based on our results, we reason on the decomposition of other design patterns in Section 8.3.

8.1 Limitations of Family Role Models

We proposed family role models in Section 3.3 as a means to describe collaborations of features without describing a definite feature model. According to Alves et al. [1], for a set of features, different feature model representations exist that are semantically equivalent. In order to model such semantically equivalent feature model representations in a unified way and to address the general collaborations of features instead of a definite design, FRMs can be applied. In FRMs, we can express the following conditions between two feature roles:

Equivalence A feature should play two feature roles

Requires A feature role requires another feature role

Prohibition A feature cannot play two feature roles

Consequently, we cannot address constraints concerning the definite design, for instance, whether a feature should be mandatory or optional. Also, we cannot express the encapsulation of features within a feature group or enforce a parent-child relationship between features. We can only address general collaborations of feature roles that are played by features, whereby we can constrain the feature model only to a limited extent.

When applying FRMs to model variability-aware design patterns, we express conditions that have to hold *assuming* the design pattern is applied in a respective product. Hence, if all conditions of the FRM hold, the design pattern is applied in the product in the suggested, decomposed fashion. However, if we would allow to make parts of the design pattern mandatory using an FRM, we would limit the variability of the SPL to always contain the design pattern. Constraining feature groups or parent-child relations would constrain a respective feature model to this exact structure, limiting the modeling possibilities. Because we cannot assume the existence of a design pattern in every possible configuration, we cannot apply such constraints while describing the general decomposition of a pattern. Hence, we cannot express constraints on the whole configuration space, but only on a subset of the configuration space that explicitly contains the design pattern.

8.2 Comparison to Previous Work

In our case study, we observed differences compared to the case study results of our prior work (cf. Section 2.5) [37]. In both case studies, we applied automated design pattern detection in order to detect a set of design patterns across several SPLs. However, compared to our prior case study, we completely revised our approach and we selected a different set of patterns. We observed the following differences.

First, in this work, we only detected one single match for the strategy pattern (cf Table 6.2), while, in prior work, we detected an overall number of 81 strategy pattern instances, of which 18 are of a decomposed nature (cf. Table 2.3). This huge gap is caused by the higher accuracy of our new detection technique that results from the more convenient system representation (cf. Chapter 4) as well as the query-based detection (cf. Chapter 5), which eases formulating and executing conditions. Moreover, we formulated precise conditions for the elimination of false positives based not only on the structure, but in contrast to the prior case study, also on the intent of the respected design pattern (cf. Section 6.3). Consequently, the results of our prior case study concerning the strategy pattern were wrong.

Secondly, we increased the number of matches for the observer pattern, however, this increase is caused by analyzing the new SPL *FeatureAMP*, which did not exist during our prior work.

As a third, major difference, we observed a lack of use of the *feature template method (FTM)* (cf. Section 2.5), a feature-oriented design pattern, which is similar to the template method pattern. We discovered the FTM in previous work [36] and observed a significant application of the FTM in combination with other design patterns, such as the visitor pattern [37]. For example, the FTM is applied to introduce the concrete visitor classes to a client in order to make them available. However, in this work, we did not detect a regular application of the FTM for introducing concrete subclasses of design patterns. Only a few concrete observers in *FeatureAMP* are introduced using an FTM, however, most of them are encapsulated in larger components, such as a *playlist*, which themselves are introduced using the FTM pattern. Hence, there is no direct, but only transitive combination of both patterns. The same concept is used for the template method pattern. In *Violet*, for example, concrete implementations of the template method are comprised by larger components (e.g., the `PackageNode` in feature *ClassDiagram* is contained by a `ClassDiagramGraph`). These larger components are then added to the software using an FTM, for instance, in case of *Violet*, the `main` method of the `UMLEditor` class. In order to gain more insights on the application of the FTM, detailed analysis of its occurrences and application is necessary.

8.3 Reasoning on Variability-Aware Application of Other Design Patterns

Based on our results in Section 6.4 and the suggestions we derived for the analyzed design patterns in Chapter 7, we can reason about the decomposition of other design patterns that appear to be suitable for decomposition in the context of FOP. Usually, only the concrete parts of design patterns appear to be decomposed. We use this assumption in the following to reason about possible decompositions of other design patterns. However, in order to suggest actual guidelines or application rules, an analysis of their application with empirical evaluation is required.

The Adapter pattern might be decomposed by introducing an adaptee in a single features combined with its corresponding adapter. Moreover, an adapter might be introduced in a subsequent feature to the adaptee in order to change the adaptee's behavior without refining it. However, the latter would require to change existing references to the adaptee to target the adapter, which could emerge in the feature-optionality problem.

The Decorator pattern might be decomposed by introducing different decorators in different features, which can then be dynamically added extending the behavior. A similar approach is applied by Rosenmüller et al. [34] in order to realize dynamic composition of feature-oriented SPLs.

The Facade pattern might be applied in order to hide a whole subsystem introduced in a feature. Such a solution would relate to the idea of feature interfaces [22].

The Mediator pattern might be decomposed by providing different colleagues in different features. However, the mediator must be made aware of the existing colleagues.

The Proxy pattern might be decomposed similarly to the adapter pattern by either encapsulating both, real subject and proxy in the same feature, or introducing the proxy in a subsequent feature to the real subject.

The Abstract Factory pattern might be decomposed by encapsulating products together with their factories. This way, we would modularize product-specific behavior.

The Bridge pattern might be decomposed by encapsulating concrete subclasses of both, abstraction and implementor. Because the bridge pattern decouples an abstraction and its varying implementation, there should not be any dependency between both. Hence, we argue, the bridge pattern might resolve a feature-optionality problem of an abstraction and its implementation.

The Chain of Responsibility pattern might be decomposed by introducing different handlers in different features, however introducing several optional handlers to the same chain of handlers could result in the feature-optionality problem.

The Visitor pattern might be decomposed by introducing both, concrete visitors and concrete elements, in single features. Based on the results of prior work [37], such a decomposition is sensible. However, if both, elements and visitors, are optional, the feature-optionality problem could emerge.

The Extension Objects pattern might be decomposed by introducing different concrete extensions in single features. Additionally registering extension objects at the corresponding concrete subjects is crucial. This dependency could lead to the feature-optionality problem if both, concrete extensions and concrete subjects are introduced in single, optional features.

The Role Object pattern might be decomposed similarly to the extension objects pattern by decomposing concrete roles along features.

9 Conclusion, Related and Future Work

In this chapter, we finish this thesis by providing a brief conclusion, after which we state related work. Finally, we provide ideas for future work.

9.1 Conclusion

In this thesis, we introduced the idea of variability-aware design patterns, i.e., design patterns that are applied in a decomposed fashion in a modular SPL implementation approach, such as feature-oriented programming. To this end, we created FOPJaMoPP, a variability-aware system representation for Java-based feature-oriented SPLs. Moreover, we developed Pattern Demon, an automated, family-based design pattern detection technique based on FOPJaMoPP. In order to derive and describe guidelines and application rules for variability-aware design patterns, we introduced family role models as an extension to role modeling. Using family role models, we can express a mapping of design pattern roles to features and describe feature collaborations in a role-based fashion.

Using FOPJaMoPP and Pattern Demon as well as family role models, we revealed the variability-aware application of certain design patterns, *composite*, *observer*, *objectifier*, *strategy* and *template method*, by conducting a case study on several existing feature-oriented SPLs, where we searched for implementations of design patterns. Using the results, we derived guidelines and application rules for variability-aware design patterns, which we documented in a pattern catalog.

We mainly observed that abstract parts of design patterns, which introduce the general concept of the pattern, are always introduced within one single feature. In contrast, concrete parts of design patterns, such as concrete implementations of observers or strategies, are often decomposed along features. Moreover, for the *observer* and *template method* patterns, features introducing concrete parts are distributed across the whole feature model, whereas, for the *strategy* (and *objectifier*) pattern, concrete parts are introduced together with the abstract parts within a feature group. The *composite* pattern, which does not usually introduce specific concrete subclasses, does not occur in a decomposed fashion. However, these findings are limited by the small set of analyzed design patterns as well as the small set of analyzed SPLs.

In conclusion, a decomposition of specific design patterns appears to increase modularity and reuse by decoupling specific implementations from abstraction. Moreover, fine-grained customization is allowed by exploiting runtime variability offered by design patterns. However, when decomposing design patterns, feature interactions and especially the feature-optionality problem (cf. Section 2.2.3) need to be taken into account.

9.2 Related Work

We address related work separately for FOPJaMoPP, Pattern Demon and the overall approach of variability-aware design patterns in the context of FOP.

9.2.1 FOPJaMoPP

With FOPJaMoPP, we created a parser and reference resolver for feature-oriented Java code by extending JaMoPP [18] (cf. Chapter 4). The result is a 150% AST of all feature-oriented roles containing resolved inter-feature references to all existing declarations of classifiers, fields and methods.

Fuji [25] is fully-fledged compiler for feature-oriented Java code developed by Kolesnikov and based one the general compositional approach of FeatureHouse [5]. Because Fuji does not check the validity of the configuration, it allows to compose a product that contains all features. Moreover, in the resulting AST, Fuji annotates which element is introduced in which feature. This way, Fuji can also be applied for family-based analysis. However, with FOPJaMoPP, we parse mere feature-oriented code without composing it, leading to a modular representation in the 150% AST, which is more appropriate for family-based analysis.

Kästner et al. [24] developed the variability-aware parser and type checker TypeChef, which takes lexical macros and conditional compilation (e.g., `#ifdef`'s) into account, in order to parse and analyze SPLs developed using annotative approaches. With FOPJaMoPP, we address a similar problem, however, in the context of modular programming approaches, such as FOP, where we face different challenges, such as resolving inter-feature references.

9.2.2 Pattern Demon

In previous work [37], we developed a static design pattern detection technique completely based on the pattern detection for object-oriented software by Heuzeroth et al. [20], which we applied to the Fuji AST. Pattern Demon is based on our prior technique, however, with FOPJaMoPP, we employed a new variability-aware system representation. Moreover, we revised the detection technique to use graph-pattern matching. We also applied role modeling to describe the general characteristics of a design pattern in order to derive a set of structural as well as data and control flow conditions that we incrementally check on a 150% AST of a feature-oriented SPL.

A variety of approaches exist to detect design patterns in object-oriented software [13, 29]. With Pattern Demon, we contribute to this field. We reuse the AST-based concepts of Heuzeroth et al. [20] as well as the idea of graph-based detection of Tsantalis et al. [40]. We extend the field of automatic design pattern detection by proposing a family-based approach for feature-oriented SPLs. Moreover, we introduce role modeling to the field of pattern detection as a means to describe static characteristics of design patterns.

9.2.3 Variability-Aware Design Patterns

We contribute to the research on design and modularity in feature-oriented programming [3, 22] by proposing the idea of variability-aware design patterns as a means to exploit both, object-oriented and feature-oriented, modularity. Also, Hannemann et al. [17] developed and compared Java and AspectJ implementations of all 23 GoF patterns, where they modularized the patterns across aspects.

9.3 Future Work

During the course of this thesis, various aspects arose that can be taken into account for future work on the topic of design patterns in the context of SPLs. Most importantly, in order to derive more detailed guidelines and application rules, more design patterns should be analyzed since more design patterns (cf. Section 8.3) appear to be suitable for modularity and variability in the context of SPLs. Moreover, more feature-oriented SPLs have to be analyzed, since our results are based on a limited set of existing SPLs. Analyzing more SPLs would increase the chance of matches and cover more existing applications of design patterns, hence, produce more data to derive guidelines from.

On a more technical level, the design pattern detection technique should be improved in order to allow a better analysis of design patterns in FOP. An increase in accuracy could be achieved by eliminating more false positives and duplicates automatically. To this end, maybe naming conventions or more fine-grained data and control flow analyses as well as automated semantic checks might be applicable. Detecting more variations of design patterns is also not straightforward. Since we are using static analyses, detecting more pattern variants usually results in a higher number of false positives. Nevertheless, we argue, sample-based analysis on selected products using dynamic pattern detection approaches might be a way of improving the detection to find variations of design patterns.

An issue with FOPJaMoPP arose when an `original`-call was detected as the call to a hook method of a *template method* pattern. To eliminate such issues, we should parse `original`-calls not as regular method calls targeting all declarations of the enclosing method, but rather create a new, distinguishable representation for `original`-calls in the metamodel and adapt the parser accordingly.

Furthermore, as we implied in Section 8.1, reasoning on the extent of family role models should be performed since the expressiveness, so far, is limited. However, as a means to describe general collaborations of features contributing to the implementations of a design pattern instead of a definite feature model, increasing the expressiveness of family role models could lead to a restriction of the variability space in general.

Our results on the impact of design patterns on feature interactions are not yet satisfying, however promising. More analysis on feature interactions in the context of applying design patterns in SPLs should be performed in order to derive guidelines on how to avoid unwanted, or create desired feature interactions using design patterns.

Bibliography

[1] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. "Refactoring Product Lines". In: *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. GPCE '06. Portland, Oregon, USA: ACM, 2006, pp. 201–210.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.

[3] S. Apel and D. Beyer. "Feature Cohesion in Software Product Lines: An Exploratory Study". In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE '11. IEEE, 05/2011, pp. 421–430.

[4] S. Apel and C. Kästner. "An Overview of Feature-Oriented Software Development." In: *Journal of Object Technology* 8.5 (2009), pp. 49–84.

[5] S. Apel, C. Kastner, and C. Lengauer. "FEATUREHOUSE: Language-independent, automated software composition". In: *Proceedings of the 31st International Conference on Software Engineering*. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 221–231.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement". In: *IEEE Transactions on Software Engineering* 30.6 (06/2004), pp. 355–371.

[7] D. Batory. "Feature Models, Grammars, and Propositional Formulas". In: *Software Product Lines*. Ed. by H. Obbink and K. Pohl. Vol. 3714. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2005, pp. 7–20.

[8] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. "The Role Object Pattern". In: *Washington University Dept. of Computer Science*. 1998.

[9] K. Beck, R. Crocker, G. Meszaros, J. Vlissides, J. O. Coplien, L. Dominick, and F. Paulisch. "Industrial experience with design patterns". In: *Proceedings of the 18th international conference on Software engineering*. IEEE Computer Society. 1996, pp. 103–114.

[10] G. Booch. *Object Oriented Analysis & Design with Application*. Pearson Education India, 2006.

[11] P. Clements and L. Northrop. *Software Product Lines – Practices and Patterns*. Addison-Wesley, 2001.

[12] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[13] J. Dong, Y. Zhao, and T. Peng. "A review of design pattern mining techniques". In: *International Journal of Software Engineering and Knowledge Engineering* 19.06 (2009), pp. 823–855.

[14] E. Freeman, E. Freeman, B. Bates, and K. Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004.

[15] E. Gamma. "The extension objects pattern". In: *Proceedings of the 1996 Conference on Pattern Languages of Programs*. PLoP '96. 1996.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1994.

[17] J. Hannemann and G. Kiczales. "Design pattern implementation in Java and aspectJ". In: *ACM SIGPLAN Notices* 37.11 (2002), pp. 161–173.

[18] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. "Closing the Gap between Modelling and Java". In: *Software Language Engineering*. Ed. by M. van den Brand, D. Gašević, and J. Gray. Vol. 5969. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009, pp. 374–383.

[19] S. Herrmann. "Object Teams: Improving Modularity for Crosscutting Collaborations". In: *Objects, Components, Architectures, Services, and Applications for a Networked World*. Ed. by M. Aksit, M. Mezini, and R. Unland. Vol. 2591. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003, pp. 248–264.

[20] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. "Automatic design pattern detection". In: *Proceedings of the 11th IEEE International Workshop on Program Comprehension*. IWPC '03. 05/2003, pp. 94–103.

[21] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-oriented domain analysis (FODA) feasibility study*. Tech. rep. DTIC Document, 1990.

[22] C. Kästner, S. Apel, and K. Ostermann. "The Road to Feature Modularity?" In: *Proceedings of the 15th International Software Product Line Conference, Volume 2*. SPLC '11. Munich, Germany: ACM, 2011, 5:1–5:8.

[23] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. "On the impact of the optional feature problem: analysis and case studies". In: *Proceedings of the 13th International Software Product Line Conference*. SPLC '09. San Francisco, California: Carnegie Mellon University, 2009, pp. 181–190.

[24] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. "Variability-aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 805–824.

[25] S. Kolesnikov. "An Extensible Compiler for Feature-Oriented Programming in Java". Master's Thesis. Passau University, Department of Informatics and Mathematics, 02/21/2011.

[26] J. Liu, D. Batory, and C. Lengauer. "Feature oriented refactoring of legacy applications". In: *Proceedings of the 28th International Conference on Software Engineering*. ICSE '06. Shanghai, China: ACM, 2006, pp. 112–121.

[27] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[28] C. Prehofer. "Feature-Oriented Programming: A Fresh Look At Objects". In: *Proceedings of the 11th European Conference on Object-Oriented Programming*. ECOOP '97. Springer, 1997, pp. 419–443.

[29] G. Rasool and D. Streitferdt. "A Survey on design pattern recovery techniques". In: *IJCSI International Journal of Computer Science Issues* 8.2 (2011), pp. 251–260.

[30] T. Reenskaug, P. Wold, and O. A. Lehne. *Working with objects: the OOram software engineering method*. Manning Greenwich, 1996.

[31] D. Riehle. "Describing and composing patterns using role diagrams". In: *White Object-oriented Nights: Proceedings of the First International Conference on Object-Oriented Technology*. Vol. 96. WOON 1. 1996.

[32] D. Riehle. *A role-based design pattern catalog of atomic and composite patterns structured by pattern purpose*. Tech. rep. Ubilab Technical Report 97.1. 1. Zürich, Switzerland: Union Bank of Switzerland, 1997.

[33] D. Riehle and T. Gross. "Role Model Based Framework Design and Integration". In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. Vancouver, British Columbia, Canada: ACM, 1998, pp. 117–133.

[34] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. "Code Generation to Support Static and Dynamic Composition of Software Product Lines". In: *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*. GPCE '08. Nashville, TN, USA: ACM, 2008, pp. 3–12.

[35] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Software Product Lines: Going Beyond*. Ed. by J. Bosch and J. Lee. Vol. 6287. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 77–91.

[36] S. Schuster and S. Schulze. "Object-oriented Design in Feature-oriented Programming". In: *Proceedings of the 4th International Workshop on Feature-Oriented Software Development*. FOSD '12. Dresden, Germany: ACM, 2012, pp. 25–28.

[37] S. Schuster, S. Schulze, and I. Schaefer. "Structural Feature Interaction Patterns: Case Studies and Guidelines". In: *Proceedings of the Eighth International Workshop on Variability Modelling of Software-Intensive Systems*. VaMoS '14. Sophia Antipolis, France: ACM, 2013, 14:1–14:8.

[38] N. Shi and R. Olsson. "Reverse Engineering of Design Patterns from Java Source Code". In: *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. ASE '06. 09/2006, pp. 123–134.

[39] Y. Smaragdakis and D. Batory. "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs". In: *ACM Transactions on Software Engineering and Methodology* 11.2 (04/2002), pp. 215–255.

[40] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. "Design Pattern Detection Using Similarity Scoring". In: *Software Engineering, IEEE Transactions on* 32.11 (11/2006), pp. 896–909.

[41] M. Völter, T. Stahl, J. Bettin, A. Haase, and S. Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[42] W. Zimmer. "Relationships between design patterns". In: *Pattern languages of program design* 1 (1995), pp. 345–364.

Eidesstattliche Erklärung

Ich versichere, dass ich die beiliegende Masterarbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Ort, Datum Unterschrift

Technische Universität Carolo-Wilhelmina zu Braunschweig
Institut für Softwaretechnik und Fahrzeuginformatik

Mühlenpfordtstr. 23
D-38106 Braunschweig